**The Technical University of Catalunya**

# Job-Guided Scheduling Strategies for Multi-Site HPC Infrastructures

**PhD**

Francesc Guim Bernat

Computer Scientist by The Technical University of Catalunya

2008

**Computer Department Architecture**

Technical University of Catalunya

# Job-Guided Scheduling Strategies for Multi-Site HPC Infrastructures

PhD

**Author**: Francesc Guim Bernat
*Computer Scientist by The Technical University of Catalunya*

**Advisor**: Dr. Julita Corbalan Gonzalez
*Dr. Computer Science*

2008

Title:
Job-Guided Scheduling Strategies for Multi-Site HPC Infrastructures

Author:
Francesc Guim Bernat

**Thesis Committee**:

| | | |
|---|---|---|
| President | : | PRESIDENT |
| Speakers | : | VOCAL1 |
| | | VOCAL 2 |
| | | VOCAL2 |
| Secretary | : | SECRETARI |
| Surrogate Decision Makers | : | SUPLENT1 |
| | | SUPLENT2 |

Agree to mark with the qualification:

Barcelona, XX de XXXX de 2008

*Acknowledgments*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Abstract

From the early eighties HPC architectures have evolved from single processor machines to very sophisticated architectures such as multi-cluster systems composed of heterogeneous nodes. Commonly, access to these systems has been controlled by batch systems which schedule and manage the jobs that users submit. As is shown in figure 1.1, these scheduling systems (henceforth referred to as ”local scheduling scenarios”) are composed of two different scheduling elements: the local scheduler that decides which jobs have to run when, and the local resource manager, that decides where the jobs have to run.

This thesis targets local scenarios with characteristics that are representative of most HPC centers. As far as the architecture is concerned, we assume clustered architectures which can be used to model a wide variety of systems. We consider scheduling strategies that are space-sharing (the system is partitioned and jobs run in dedicated partitions) and run-to-completion (jobs are not preempted during their execution). An example of these scheduling policies is backfilling strategies which we describe in the following chapter. Furthermore, we assume that jobs submitted to the systems are rigid jobs (the number of processors used is fixed at submission time).

In the first part of the Thesis, we focus on improving the performance of local scheduler scenarios from two perspectives: (1) We analyze the use of job runtime predictors models by local schedulers rather than user runtime estimations, and (2) we propose new models, mechanisms, and policies that involve both the local scheduler and the local resource manager.

When users want to submit their HPC applications, they have to provide the scheduler with a description of the job that they want to be executed, and a list of resource requirements. Usually, these requirements include the requested number of processors, the runtime estimation, and other resource requirements, such as the amount of disk required or the memory required by the application.

The job runtime estimation is required by job backfilling policies, which are a set of policies that base their scheduling decisions on the estimated runtime of the job and the number of requested processors. In many situations the user does not have enough experience or knowledge to estimate how long the jobs will run. Since backfilling policies are used in most of the HPC systems, we will use them as our starting point, as far as job scheduling is concerned.

Recent papers have proposed various ways of using prediction techniques rather than user estimates in backfilling policies. They have also provided results about how specific predictors performed with a set of workloads and backfilling policies. In the first part of the Thesis we present a set of models which characterize the behavior that prediction techniques have shown in HPC centers. We define a set of prediction models which have several properties inherent to the predictors' estimations, and that impact on the performance of the system. The models are designed to evaluate scheduling policies which use predictions rather than user estimates. This study provides ways to evaluate the impact of prediction errors in the performance of prediction based scheduling policies. Moreover, our models are evaluated using the most representative backfilling policies, and the study demonstrates that these techniques could be used in real systems using the evaluated variants without causing any negative effects on the performance of the system. We also identify the types of prediction errors which must be avoided when using predictions in these systems.

There is a huge research background to evaluating the performance of backfilling strategies. Usually, researchers use simulation models which evaluate how a given scheduling policy behaves with different workloads inputs. A key point in studying the performance of these scheduling policies is the resource models used in these evaluations. All the papers that have been presented in this area are based on the use of one-dimensional resource models. Thus, they only model the amount of free resources which are free and used during the simulation: they do not model where the different jobs are allocated. Furthermore, they do not consider how resources such as memory bandwidth or network bandwidth are used by the different allocated jobs. However, in real systems the performance of such policies may substantially differ from theoretical studies, since resource usage has a real impact on the performance of the system. Thus, depending on how the jobs are mapped to the processors of the host, performance can vary. As a result, the conclusions obtained using traditional models cannot be applicable to all real systems.

Taking all these observations into account, we describe a new resource simulation

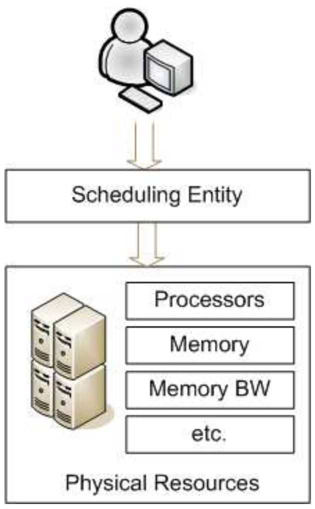


Figure 1.1: The local scheduler scenario

model that can be used in the evaluation of scheduling policies in HPC systems where job resource usage is considered. We present a bi-dimensional model which considers how different jobs processes are mapped to the physical resources during the simulation. We demonstrate how this model provides quantitative and qualitative differences in the evaluations of the policies in the HPC architectures, with respect to the models used in traditional studies. Furthermore, based on these experiences, we present new job scheduling strategies which use the proposed model whose objective is to optimize the job response time by considering it resource utilization of the systems. We present two new resource selection policies and one new job scheduling policy.

In the second part of the Thesis we examine how to optimize scheduling architectures composed of multiple heterogeneous local scheduler scenarios (henceforth referred to as ”meta-scheduling scenarios”). The increasing complexity of local systems has led to new distributed architectures which are composed of multiple computational resources with different characteristics and policies (see Figure 1.2). Systems composed of several computational resources provided by different Research centers and Universities have become very popular. Job scheduling policies have been adapted to these new scenarios in which several independent resources have to be managed. In these new scenarios, traditional scheduling techniques have evolved into more complex and sophisticated approaches where other factors, such the heterogeneity of the resources or the geographical distribution, have been taken into account.

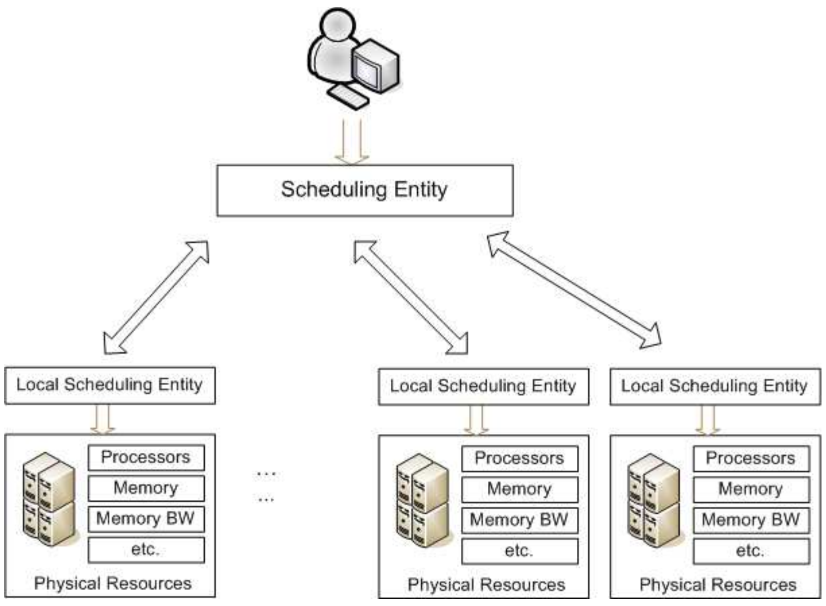


Figure 1.2: The meta-scheduler scenario

Several centralized scheduling solutions have been proposed in the literature for these environments, such as centralized schedulers, centralized queues and global controllers. These approaches use a unique scheduling entity responsible for scheduling all the jobs that are submitted by users. In general, as with local scenarios, the user has to contact this centralized entity in order to submit her/his jobs, and specifying his/her requirements (user runtime, the number of processors etc.). This global scheduler decides which centers these jobs are finally submitted to. Hence, when the global scheduler chooses a local scheduler scenario to submit the job to, it contacts the local scheduler, provides the job description and requires the local scheduler to schedule and finally execute the job. In the second part of this Thesis we present a non-centralized scheduling designed to optimize the service that users receive in such architectures.

We propose substituting the use of a global scheduler or global structures to manage jobs submitted in these scenarios with self-scheduling techniques which dispatch jobs submitted to the set of distributed centers. In this architecture, the jobs are scheduled by their own dispatcher (ISIS-Dispatcher) and there are no centralized scheduling decisions. The dispatcher is aware of the status of the different resources that are available for the job, but it is not aware of the rest of the jobs that other users have submitted to the system. Thus, the job itself decides which is the most appropriate local scheduler for its execution. The target architecture of our work is distributed systems where each computational resource is managed by an independent scheduler (such as MOAB or SLURM). The core of the ISIS-Dispatcher algorithm is based on task selection policies. We have designed three new selection policies that fit our architecture and that use prediction techniques which are designed to optimize job wait time, job slowdown and the economic cost related to job execution.

# Chapter 2

# Introduction

## 2.1 The Local Scheduler scenarios

High Performance Computing Centers (HPC) provide computational resources that allow users to execute their scientific applications. An example is the Basic Local Alignment Search Tool (BLAST) application (2). It implements an algorithm to compare primary biological sequence information, such as the amino-acid sequences of different proteins, or the nucleotides of DNA sequences. Another well known example is the Car-Parrinello Molecular Dynamics, better known as CPMD (44), which is a package for performing ab-initio quantum mechanical molecular dynamics.

These applications require access to resources with high computational power: parallel processing and capabilities for accessing and processing data during their executions. To fulfill these requirements the computing machines possess advanced features and technologies such as the General Parallel File System (GPFS (85)), a high-performance shared-disk clustered file system, or the Myrinet (7) networking system, a high-speed local area networking system designed to be used as an interconnection between multiple machines. In addition to providing such hardware resources, the centers also have to provide software functionalities to run the HPC applications. For example, many of them may require the availability of message passing interface libraries (MPI), such as the OpenMPI (49)(64) or MPICH (52) libraries, or libraries for shared memory multiprocessing programming such as OpenMP (3).

Access to HPC resources are controlled by two different software components: the Job Scheduler and the Local Resources Manager. Figure 2.1 provides a general overview of the different elements that are involved in a HPC system and their relations. As can be observed, these computational resources are composed of a set of physical resources (the

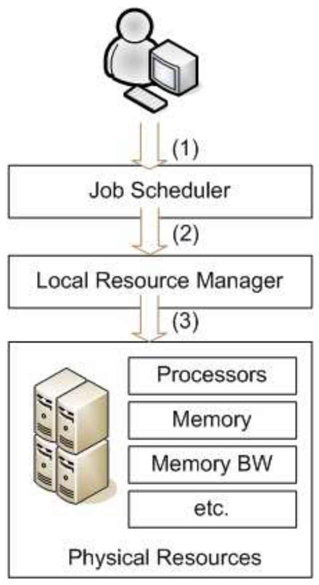


Figure 2.1: The local scheduler internals

processors, the memory, the I/O system etc.) that are managed by the local scheduler and the local resource manager (LRM). The local scheduler is responsible for scheduling the jobs that users submit and the local resource manager is responsible for controlling access to the physical resources.

The figure also describes the scheduling stages the jobs go through. In the first step *(1)* the user submits the job to the local scheduler. In the second step *(2)*, once the job is queued in the scheduler, following a Job Scheduling Policy it decides which of the queued jobs should start. In the last step *(3)*, once a job has been chosen to start, the scheduler requires the local resource manager to physically allocate the job processes to the processors. The Local Resource Manger (LRM), following a given resource selection policy, chooses the physical processors where the job has to run and allocates them accordingly. Some commercial systems such, as MAUI (83) or MOAB (84), integrate resource selection policy into part of the local scheduler's responsibility, limiting the role of the LRM to that of a process manager that forks and controls processes.

The process of selecting which jobs start when, and of deciding which processors carry out the task, may seem quite simple, but it is, in fact, a very complex task that has been the focus of much research in the last few decades. These policies have to find mechanisms which can balance the requirements of the resource owners with the requirements of the users. On one hand, the owners of the resources require the maximum utilization of their resources. From their point of view the goal of scheduling architectures is to maximize the throughput of the system [1] and to minimize the idle time of processors. On the other hand, the users require their jobs to be started and finished as soon as possible. Thus, there are several factors such as fairness and starvation that must be taken into account when designing and evaluating these policies.

Not all the requirements that the policies have to accomplish are functional requirements. In HPC architectures several factors are crucial when designing scheduling strategies. Firstly, policies must take into account the structure and characteristics of the targeted architecture. They must take into account factors such as machine partitions (i.e.: the system can be split into different logical partitions), the available resources (such as the number of processors, the amount of memory, the amount of disk, etc.), the architecture of the system (e.g. a cluster based architecture) or how the resources have to be used (e.g. jobs are scheduled in a shared mode and use a run-to-completion approach).

Another issue that has to be considered by the scheduling strategies is the information that is available at submission time. Certainly, the amount and accuracy of information may determine the performance that the scheduler can achieve with its scheduling decisions. In the current systems, this information comes from the data that the user provides at submission time (e.g. the estimated runtime of the job or the estimated memory usage) and the data that is available for the scheduler regarding the status of the system (e.g. the amount of memory consumed, or the load of a given node). The amount of available information available at job submission time is important to the design of sophisticated policies, but even more important is the quality of these inputs, since it is the quality of inputs that determines the success or failure of a given scheduling policy.

Backfilling policies are the most widely used in high performance centers. These policies base scheduling on the job runtime estimation provided by the user at submission time. As will be discussed later, backfilling policies are an optimization of the First Come First Served scheduling policy; they backfill the jobs that are queued after the head of the queue, if the estimated start time of this last one is not delayed.

In this Thesis we focus on scheduling architectures where the jobs that users submit are rigid jobs (74)(1). In this type of jobs the number of processors requested is fixed at submission time.

[1] The throughput could be measured by the number of jobs finished per unit of time.

## 2.2 The Meta-Scheduler scenarios

In the previous section we discussed the scheduling components that are involved in local scheduler scenarios. We also explained their functionalities, their relations among themselves, and the requirements that the users and owners make on them. Finally, we analyzed other factors that these scenarios have to take into account when administrating computational resources.

Although the local scheduler scenario is currently prevalent in HPC infrastructures, forthcoming HPC architectures are composed of different computing machines owned by one or more different centers or organizations. As is shown in figure 2.2, these scenarios allow several local scheduler scenarios to be available in the same HPC system.

In this meta-scheduler scenario, users have potential access to many resources that may be heterogeneous. New scheduling requirements have been identified and more sophisticate scheduling approaches have been defined for these architectures. The most extended model that has been used in these topologies in recent years uses a global meta-scheduler.

The meta-scheduler is a new scheduling component that allows for the scheduling of jobs on the top of all available centers. As is shown in figure 2.2, in this centralized scheduling model, first, the user submits the job to the meta scheduler (*step (1)*) and the job is immediately queued in the meta-scheduler job queue. In the next step (*(2)*) the meta-scheduler schedules the queued jobs, deciding which center and resource they have to be submitted to. In general, the local scheduler scenarios maintain the traditional scheduling architecture presented in the previous section (the job scheduler and the local resource manager). Thus, once a job is submitted to the local scheduler, the job is scheduled by the local scheduler and the meta-scheduler has no control over how the scheduling is carried at this level. At this level, as we explained above, the scheduler decides which jobs have to start (step *(3)*) and the LRM decides which processors they have to run in(step *(4)*).

We use the term ”meta-scheduler” because these scheduling entities interact with local schedulers deployed in local scenarios. Thus, in these distributed scenarios, rather than having direct communication with local resources, the meta-scheduler has to submit to and extract information from local systems using local schedulers interfaces.

In local scheduling scenarios it is usually complex for users to provide accurate estimations about the runtime of the jobs that they submit. In these meta-scheduling scenarios, this problem becomes even harder. Here, users do not really know which center the job will be finally executed in. Furthermore, the heterogeneity of the local systems can be really high. As a result, in many situations estimating job runtime

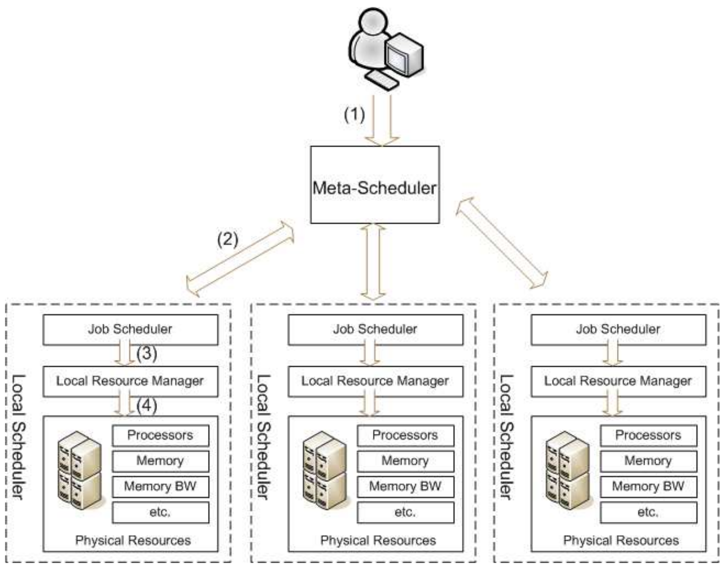


Figure 2.2: The meta-scheduler scenario

becomes an impossible task.

## 2.3 The scheduling problem

### 2.3.1 In the local scheduler scenario

Given the list of jobs that are waiting to be executed, their characterization (i.e.: the number of requested processors, the executables, the input files, etc.) and the status of the system, the job scheduling problem consists of how to decide which jobs have to start, and in which processors. This problem can be divided into two differ stages: the job scheduling policy and the resource selection policy.

As is shown in figure 2.3, in the first stage of the scheduling the *scheduler* has to decide which of the jobs that are waiting in the system (*Job 1*, *Job 2*, .., *Job N*) has to start, taking into account the resources that are currently free (*CPU 1*, *CPU 2*, *CPU 3*, *CPU 4*). The algorithm followed by the Scheduler in order to make this selection is called the *Job Scheduling Policy*. Once the appropriate job has been selected to start, the scheduler requires the *Resource Manager* to allocate the selected job (*Job i*) to appropriate processors, taking into account the resource requirements of the job (in the figure the *Job i* requires two processors). The Resource Manager will select the

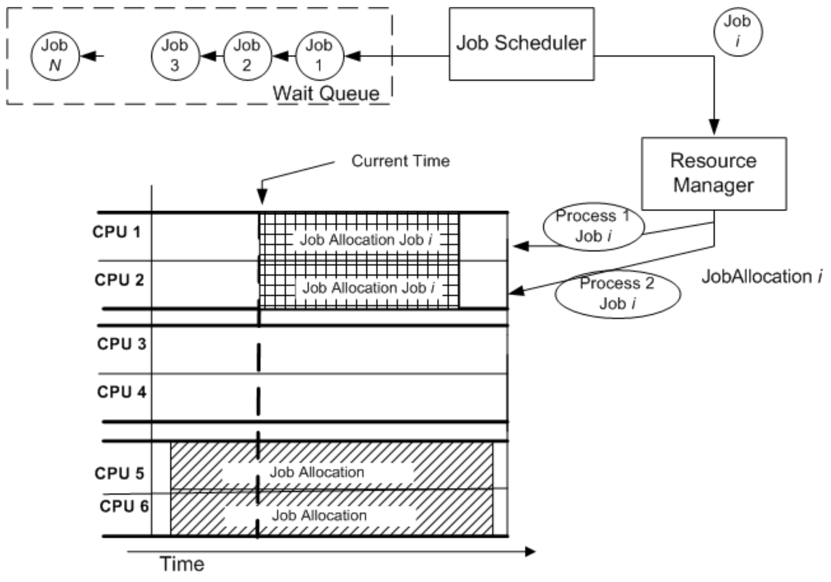


Figure 2.3: The Scheduling Problem

most appropriate processors (*CPU 1* and *CPU 2* in the example) will allocate the job processes to each of them. The algorithm followed by the Resource Manager to select the processors is called *Resource Selection Policy*.

Several scheduling strategies have been proposed in the literature to provide Job Scheduling and Resource Selection policies in order to optimize resource utilization in terms of job response times. As will be discussed below, the scheduling policies that have been most successfully deployed in most HPC Centers are backfilling policies. These policies have demonstrated that they can provide a very good balance in terms of system performance (i.e: the number of used processors) and job response time performance. However, the main problem of the algorithm used by these policies is that the user has to provide an estimation of the runtime of the jobs that are submitted, and usually, these estimations are extremely inaccurate. Furthermore, in many situations the user has insufficient skills or information to provide such information. For instance, a non computer architecture expert may not to know that his BLAST application will require 10 hours if the job that executes the allocation is done by a *Power 3* processor or 8 hours by a *Power 4* processor. Furthermore, the final runtime of the job usually depends on many different factors, such as the number of processors, status of the system, input date of the program etc. Also, in some situations, experienced users may not have enough information to estimate the runtime of the jobs they submit to a very dynamic system.

The Job Scheduling Policies are focused on the selection of the jobs that have to start, but usually, they do not consider where they will be allocated. This decision is usually taken unilaterally by the resource allocation policies. Moreover, the same resource allocation policies deployed in centers do not take into account the resource usage of scheduled jobs.

Usually, what the LRM does is to allocate jobs based on two different approaches: either a set of predefined filters are applied to the jobs' characteristics in order to find the appropriate processors, or the jobs' processes are allocated to whole nodes when a user specifies that the jobs must be allocated to a non-shared node. The first approach does not consider the status of the system, and, as a result, a memory intensive job may be allocated to a node where the memory bandwidth is almost saturated, causing a degradation of all the jobs allocated to the node. The second approach usually implies that resource utilization is substantially decreased because often the jobs do not use all of the processors and resources of the node where they are allocated.

Thus, in some architectures, when jobs share resources (i.e.: the memory of the nodes), in a given schedule the shared resources can become overloaded. This has a negative impact on the performance of the jobs that are using them, and there is a collateral negative impact on system performance. This problem is illustrated in figure 2.4. It shows two different scheduling outcomes when jobs *Job Y* and *Job X* are submitted to the system. In current scheduling strategies, the most probable outcome would be *Schedule A*. Here the jobs *Job Y* and *Job X* are allocated consecutively to the same node *Node A*. If both jobs are memory intensive, the memory bandwidth of the node becomes overloaded, and both jobs suffer a drop in their performance, which results in an increment in both of the *penalty* [2] units of time. However, if the scheduler took into account the status of the system and the resource consumption of both jobs, the most probable outcome would be that presented in *Schedule B*. In this last outcome, both Y and X jobs are not penalized.

The simulation workload models that are currently used to evaluate scheduling strategies at the local scheduler scenario do not consider how physically shared resources are used by the running jobs. We have stated that job resource usage (such as the memory bandwidth) has a direct impact on the performance of these policies. Thus, using and modeling scheduling strategies without taking into account the resource status and the resource requirements of the jobs can lead to bad scheduling decisions or wrong

[2] The penalty is the amount of time that is added to job runtime due to resource saturation. It is computed by subtracting the nominal runtime (resulting from the job execution when all the amount of the required resources are available) of the job to the final runtime of the job.

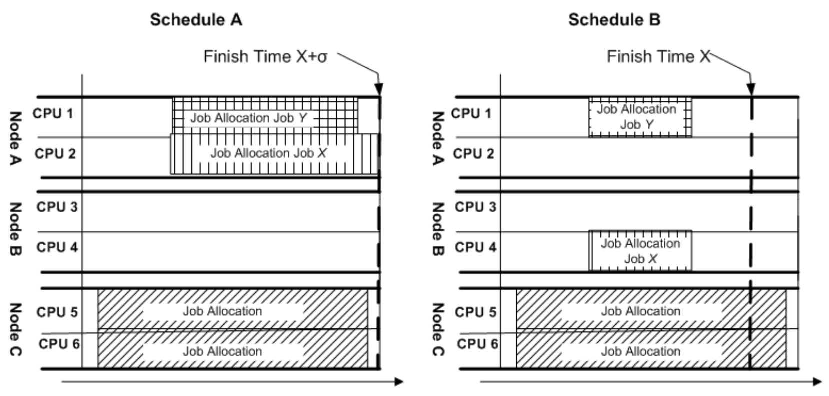


Figure 2.4: The Resource Usage Problem

conclusions in scheduling evaluations. For instance, allocating two jobs that are I/O intensive to the same node may lead to a saturation of the Ethernet bus, penalizing job runtime.

#### 2.3.1.1 The Backfilling policies

From the early nineties, local scheduling architectures and policies have been one of the main goals of research in the area of high performance computing. Backfilling policies have been deployed in the major HPC centers. A backfilling scheduling policy is an optimization of the simplest scheduling algorithm: First-Come-First-Serve (FCFS). In the FCFS the scheduler queues all the submitted jobs in arrival order and each of these jobs has an associated number of requested processors. When a job finishes its execution, if there are enough resources to start the head job of the queue, the scheduler pops the job from the queue and starts it, otherwise the scheduler has to wait until enough free processors become available and no jobs are moved to run. Figure 2.5 presents a possible outcome of a FCFS scheduling policy. The main problem of using such a policy is that the system suffers from fragmentation and the utilization of resources is really low.

A backfilling policy is an optimization of the FCFS policy. It starts jobs that have arrived later than the job at the head of the wait queue if the estimated start time of this job is not delayed. Typically, this is called a reservation for the first job. Notice that to apply this optimization the runtime of the queued jobs has to be known at submission time, since otherwise there is not enough information to assure that the start time of the head job is not delayed. Figure 2.6 presents how the outcome of the previous example could be optimized with a backfilling policy. In this example the jobs *Job 4* and *Job 5*

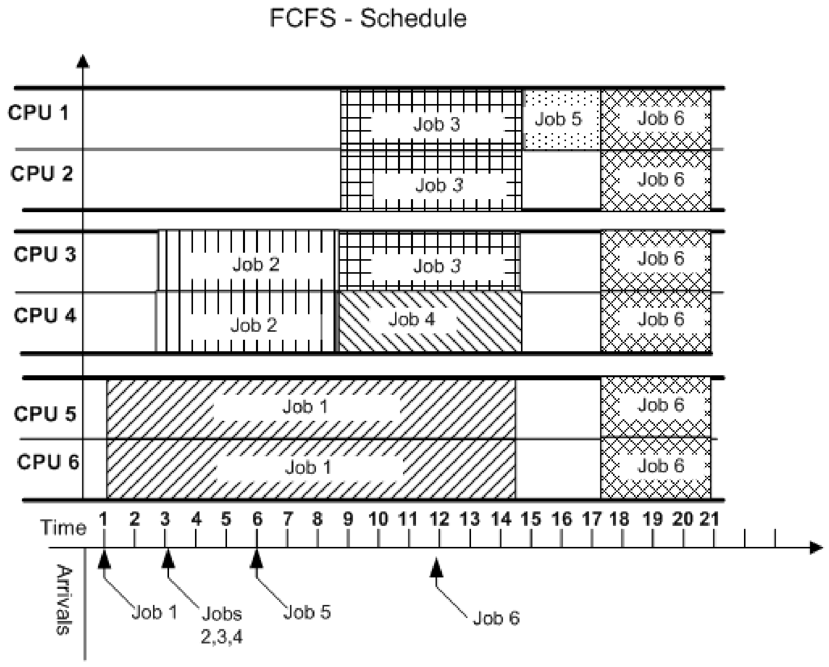


Figure 2.5: First-Come-First-Serve Scheduling

have been backfilled because the jobs *Job 3* and *Job 6* cannot start as there are not enough free processors for them. Thanks to this optimization, the utilization of the systems is improved by several orders of magnitudes. Many of the processors that would remain idle with the FCFS scheduling policy, because the head job cannot be started, are used by the backfilled jobs.

The main drawback of this algorithm is that the user has to provide at submission time the runtime estimation for the job that he/she is submitting. In situations where the runtime estimate that is provided by the user is lower than the real runtime, the job is killed by the scheduler when it detects that the job has exceeded its requested runtime. The other situation that frequently occurs in this scenario is that the estimated runtime is substantially higher than the real runtime.

In the example of figure 2.7 we present both situations. In this outcome job *Job 1* has been overestimated. Once the scheduler has detected this situation, and using the runtime estimation provided by the user, it has backfilled the job *7*. However, the job *Job 7* is later killed because it has exceeded the runtime that the user specified. Note that in the case that job *Job 7* is not killed, job *6* start time would be delayed. Hence, the

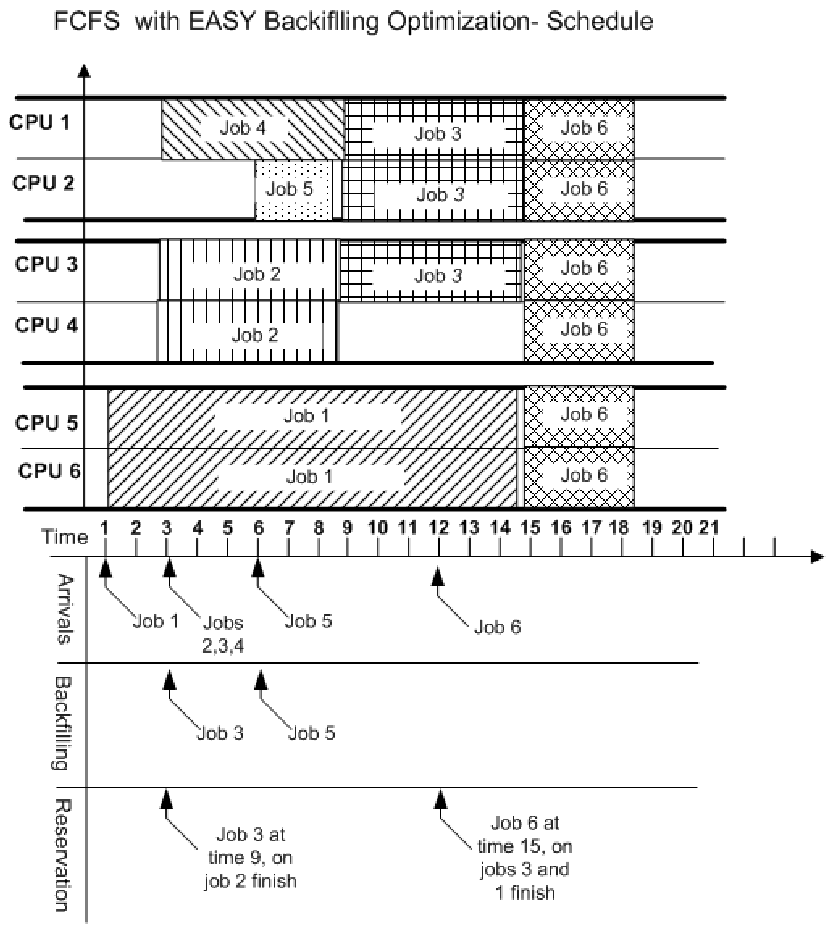


Figure 2.6: FCFS with EASY Backfilling Optimization Scheduling

main constrain of this policy would be violated. In these situations, the job is not killed, but the previous scheduling outcome may change. Many studies have tried to identify the effects of underestimations on job runtime.

The backfilling policy that we discussed in the previous paragraph is the most basic backfilling policy proposed by Lifka et al. in (88) and it is called EASY-Backfilling. Many variants of this first proposal have been described in several papers. The differences between each of them can be identified as follows:

- The order in which the jobs are backfilled from the wait queue: in the EASY variant the jobs are backfilled in arrival order, other variants have proposed

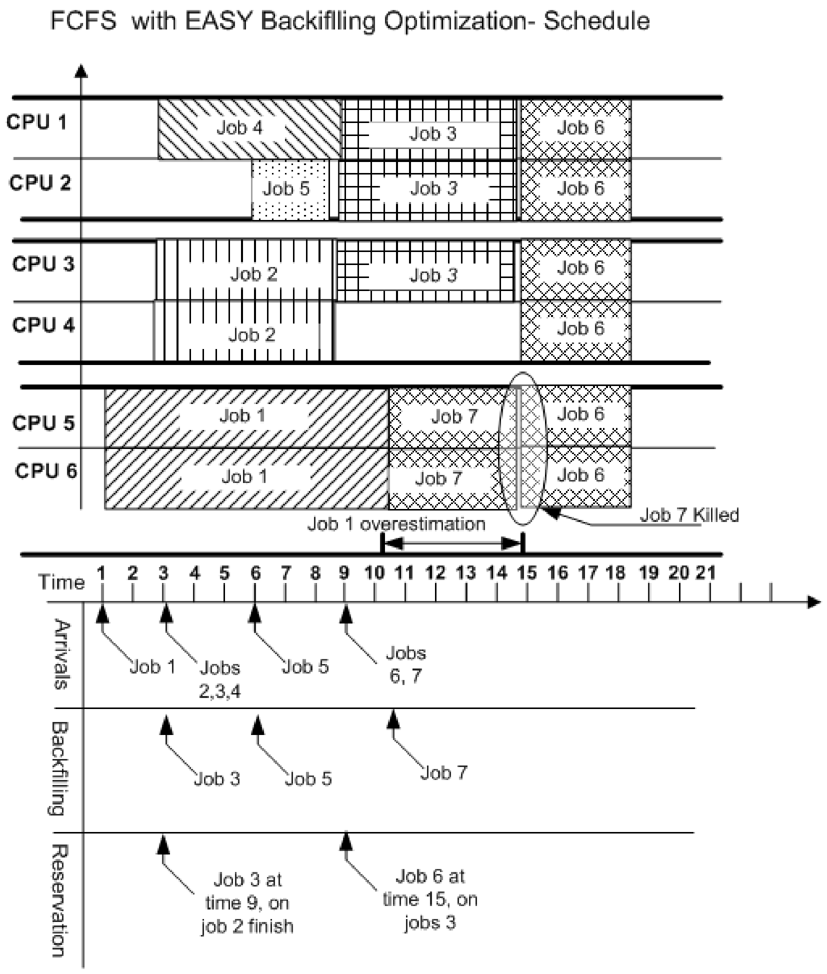


Figure 2.7: FCFS with EASY Backfilling Optimization Scheduling: Job Over/Under estimation

backfilling the jobs in shortest job first order (Shortest-Job-Backfilled-First (98)(97)). More sophisticated approaches propose dynamic backfilling priorities based on the current wait time of the job and the job size (LXWF-Backfilling (17)).

- The order in which the jobs are moved to the head of the wait queue, i.e.: which job is moved to the reservation. Similar to backfilling priorities, in the literature many papers have proposed pushing the job to the reservation in FCFS priority order or using the LXWF-Backfilling order.

- The number of reservations that the scheduler has to respect when backfilling jobs. The EASY variant is the most aggressive backfilling since the number of reservations is 1. As a result, in some situations the start time for the jobs that are queued behind the head job may experience delays due to the backfilled jobs. More conservative approaches propose that none of the queued jobs are delayed for a backfilling job. However, in practice, this last kind of variant is not usually used in real systems.

General descriptions of the most frequently used backfilling variants and parallel scheduling policies can be found in the report that Dror G. Feitelson et al. provide in (36). Moreover, a deeper description of the conservative backfilling algorithm can be found in (91), where the authors present a characterization and explain how the priorities can be used to select the appropriate job to be scheduled.

### 2.3.2 In the meta-scheduler scenario

Current HPC Infrastructures are composed of many computational resources. In these systems the user has potential access to tens of thousands of resources. In general, each of these resources are locally managed following the structure that we described in the first part of this chapter, and all the jobs are scheduled on top of these architectures by a global meta-scheduler. Thus, as shown in figure 2.8, users submit their jobs to the global meta-scheduler. The meta-scheduler schedules all the jobs that have been submitted to it following a given meta-scheduling policy. This policy, given the job descriptions and the characteristics of the resources that are available in the local centers and their status, decides where the jobs are finally submitted. Once the meta-scheduler has submitted a job to a local computational system, the job is scheduled by the local scheduler which manages the given resource.

The general picture of the scheduling architecture used in these distributed scenarios shows that they are composed of two layers of scheduling: the meta-scheduler and local schedulers. In a more detailed view we could see that these system are composed of three layers of scheduling: the two previous ones, plus the local resource manager. However, from the point of view of global scheduling, the local resource management level is not usually taken into account.

In these scenarios we can identify similar problems to the problems that we identified in the local scenarios. On one hand, the users face the problem of estimating the runtime of their jobs. While in the previous scenario this problem is difficult due to user inexperience or system dynamicity, in distributed systems the problem becomes even more difficult. Since users do not know which host their jobs will finally run in, it is very difficult for them to provide a runtime estimation.

On the other hand, the proposed meta-scheduling solutions base their decisions on the status of local resources (i.e.: load of the host) or their static properties (i.e.: number of processors). Hence, in almost cases there is no exchange of information between the meta-scheduler and the local scheduler in terms of scheduling information. Basing the selection on local resource status and capabilities may result in bad scheduling decisions. For example, the meta-scheduler may decide to submit a job to resource *Host A* because it has a higher number of free resources, however it has 20 jobs in its queue and would start earlier in a resource *Host B* that currently has less free resources, but has fewer queued jobs. As we will explain in the following subsections, we argue that better scheduling decisions can be taken when both scheduling layers interchange information related to the scheduler, rather than the status of the resources and using job oriented scheduling approaches.

The size of these systems has grown exponentially in recent years, and, as a result, another problem has appeared, namely scalability. While the older meta-scheduling solutions were acceptable for the number of hosts that where managed in the older systems, new architectures require more sophisticated scheduling strategies to provide an acceptable service to users and the centers.

## 2.4 Our Thesis

The proposals contained in our thesis aim to improve the overall scheduling decisions that are made in different High Performance Computing Architectures levels: from local resource managers to the scheduling that has to be done in complex multi-site HPC Systems. To improve the performance of the local systems and the service perceived for their user, we propose to:

- Improve resource utilization in scheduling policies, taking into account the resource requirements of jobs that are submitted, the resource characteristics of the system, and its status during the scheduling process. We propose to use job scheduling policies that, by using the functionalities and information provided by the local resource managers, avoid overloading the shared resources that are used for the allocated jobs.

- Use new resource allocation policies that minimize the saturation of shared resources used in the computational resources. Unlike the last proposal, here

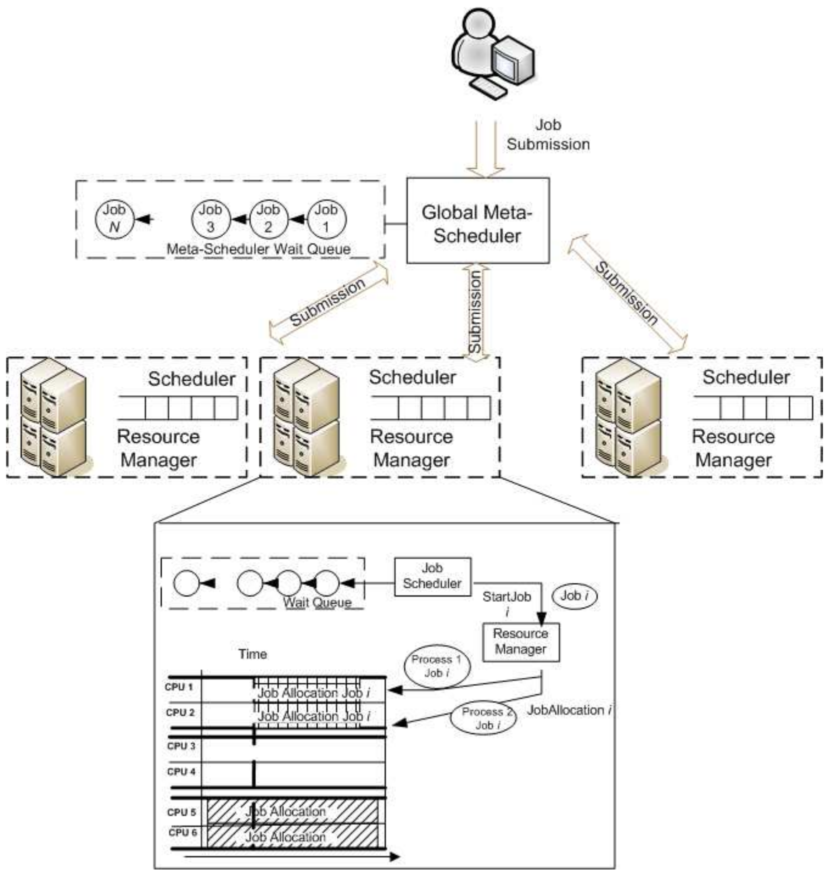


Figure 2.8: The Scheduling Problem in Distributed Architectures

the resource manager's task is to optimize resource usage for the starting jobs by making intelligent processor allocations that distribute the job processes in a way that minimizes the overloading of these shared resources and maximizes their utilization. These resource allocation policies can be used by any of the existent schedulers without the need to modify their scheduling policies.

- Use prediction techniques to estimate job requirements, such as job runtime and job resource usage characterization, and use these predictions in job scheduling policies. Thus, users are removed from the complexities of the local resources and policies and are not required to provide information in situations where accurate estimations are already provided.

To improve the performance of distributed HPC Architectures we propose to:

- Use non-centralized scheduling policies to schedule the jobs that users submit to large environments where many resources of different administrative domains are involved. Furthermore, this resource can join or unjoin at any time. Unlike traditional approaches, our proposal is to use one scheduling entity per job submitted. The objective of this entity is to optimize the performance in terms of response time that the job receives when there are different choices as to where to submit the job.

- Promote the exchange of scheduling information between local schedulers and non-centralized scheduling entities. Current approaches base decisions about where to submit jobs on the statical description of resources and on their status. However, we strongly argue that the information which global schedulers have to use in their scheduling decisions is information concerning scheduling activities and the status of the schedulers that are deployed in local architectures.

- Similar to our proposal to use at the local level, use prediction techniques to estimate job performance variables and use this information in global scheduling.

## 2.5 Contributions of this Thesis

The contributions of this Thesis are divided into three main points. The first two parts aim to provide scheduling solutions to problems in local systems which we identified in the introduction. The last part aims to provide solutions in distributed HPC distributed infrastructures.

In the first part of the Thesis we present a detailed analysis of how prediction techniques impact on backfilling based scheduling policies. We describe a set of *f* based prediction models which characterize the behaviors that prediction system may show in local scenarios, and that may impact on the performance of the scheduling policy. This analysis provides clear results and supports the argument that introducing the use of prediction in the scheduling systems is feasible and does not necessarily imply a loss of performance, or go against user interests. Furthermore, we provide a characterization of the prediction errors that show a clearly negative impact on system performance, and of errors which are not considered as crucial when predicting application run time.

In second part we provide an analysis of how the resource usage modelization of shared resources can provide, in contrast to traditional evaluation studies, relevant performance results differences when simulating backfilling-based models. Previous simulation models evaluate backfilling based policies without modeling how the resource are used once the jobs are allocated to the local resources. We present a resource usage model to evaluate resource sharing usage. Using the results obtained using this new resource usage model, we present two new resource selection policies (the **Find Less Consume Distribution**, **Find Less Consume Threshold Distribution**) that aim to minimize the job runtime penalty caused by the saturation of resources that are shared in given computational resources. Furthermore, we provide a new backfilling policy (**Resource Usage Aware Backfilling**) that cooperates with the local resource manager, and whose objective is also to minimize the saturation of shared resources. The results show that using the presented new resource allocation policies and the new job scheduling policy, performance of the system is substantially improved when taking into account the resource usage model.

In the last part of the Thesis we present the use of self-scheduling techniques to dispatch the jobs that are submitted to a set of distributed computational hosts, all managed by independent schedulers. This policy, called **ISIS-Dispatcher**, is a non-centralized and job-guided scheduling policy whose main goal is to optimize user metrics (i.e: wait time or slowdown). Scheduling decisions are made independently for each job instead of using a global policy where all jobs are considered. Additionally, as part of the proposed solution, we also demonstrate how the use of job wait time and runtime prediction techniques can substantially improve the performance of the described architecture.

## 2.6 Organization of the Thesis document

The rest of this Thesis is organized as follows:

- Chapter 3 describes the methodology that we have followed in order to evaluate all the techniques described in the Thesis. We also present the different metrics, statistical estimators and workloads that we have used in the evaluations.

- Chapter 4 describes the simulation environment that was developed in the context of this Thesis. We provide a characterization of the Alvio simulator, information about its internals and about the functionalities that it provides.

- Chapter 5 presents the prediction models that were designed to evaluate the impact of prediction errors in backfilling based scheduling policies.

- Chapter 6 describes the workload simulation model that we designed to evaluate the impact of resource sharing on scheduling strategies.

- Chapter 7 describes the resource selection policies the **Find Less Consume Distribution**, **Find Less Consume Threshold Distribution** and the Job Scheduling Policy **Resource Usage Aware Backfilling**, that aim to minimize resource saturation.

- Chapter 8 describes the use of self-scheduling techniques in HPC distributed scenarios. In this context we describe the *ISIS-Dispatching* algorithm and their evaluation in the Alvio Simulator. We describe the task assignment policies that have been designed for this architecture (*Less-WaitTime*, *Less-Slowdown* and *Less-Slowdown-Ec*).

- Chapter 9 presents our conclusions and possible future studies.

Chapter 3

# Methodology

In this Thesis we used workload simulation techniques to evaluate the performance and behaviors of all the proposed scheduling strategies. In this section we present the procedures that we followed in their evaluation and which performance metrics, statistical estimators and workloads were used as an input and output of such evaluations.

## 3.1 Simulation Procedures

The presented experiments were conducted using two C++ event-driven simulators. The first one, used for evaluating the impact of the prediction models in the backfilling policies, was implemented and used by Tsafrir et al. in the paper (98). This is a modular simulator that allows simulating the EASY-Backfilling and Shortest Job Backfilled-First policies. They are implemented extending the traditional EASY algorithm for the usage of runtime prediction rather than user estimates. The simulator also allows adding predictors modules that are used by the scheduling policies to schedule the job according its predicted runtime.

The second one, the Alvio simulator (56)(57) (described in detail in the chapter 4) was developed in the context of this Thesis. It models the different components that interact in local and distributed architectures. Conceptually, it is divided into three main parts: the simulator engine, the scheduling polices model (including the resource selection policies) and the computational resource model. A simulation allows us to simulate a given policy with a given architecture. Currently, four different policies were modeled: the First-Come-First-Served, the traditional backfilling policies, the RUA-Backfilling (introduced later), and finally, the ISIS-Dispatcher scheduling policy. For the backfilling policies the different properties of the wait queue and backfilling queue are modeled (SJF, LXWF and FCFS) and different numbers of reservations can also be specified.

The architecture model allows us to specify different kind of architectures. Currently, cluster architectures can be modeled, where the host is composed by a set of computational nodes, where each node has a set of consumable resources (currently Memory Bandwidth, Ethernet Bandwidth and Network bandwidth). The use of these consumable resources can be simulated in a high level fashion.

The procedure that we followed for evaluating each of the contributions of this Thesis is the following:

1. First, if required, extend the simulation framework to include the models or techniques to be evaluated.
2. Second, in the experiment characterization:
   (a) Select the workloads to be used in the experiments (they are described in the section 3.4)
   (b) Select the metrics and statistical estimators to be collected during the simulation (they are described in the sections 3.2 and 3.3).
   (c) Select the data collectors to be used by the simulator (see the Alvio functionalities). They are used to create the CSV files with the performance of the jobs or the status of the system during the simulation.
3. Third, decide the parameters to be evaluated in the experiments. For instance, the different numbers of prediction errors to be tested in the experiments or the scheduling policy to be evaluated.
4. Fourth, carry out the simulations and evaluate their outputs (metrics and estimators). The data created by the collectors is processed with external statistical tools such as Minitab (77), Matlab (75) or R (65).
5. Fifth, if the results of the simulation process are conclusive, the evaluation process is finished. Otherwise, the previous steps are iterated tuning the simulation parameters and models until reaching the desired point.

## 3.2 Performance Metrics

In all the evaluations we used a set of common metrics to evaluate the performance of the systems with the strategies proposed in this Thesis. The metrics that are associated to the job' performance are [1]:

1. The **wait time** variable holds the time that the job spent in the wait queue until it started to run. It is computed as:
   $WaitTime_{\alpha} = StartTime_{\alpha} - SubmitTime_{\alpha}$ Its units are in *Seconds*.

2. The **slowdown** variable for the job $\alpha$ is computed as
   $SLD_{\alpha} = \frac{RunTime_{\alpha}+WaitTime_{\alpha}}{RunTime_{\alpha}}$. This variable has no units.

3. The **bounded slowdown** of the job $\alpha$ is computed as
   $BSLD_{\alpha} = \frac{RunTime_{\alpha}+WaitTime_{\alpha}}{max(RunTime_{\alpha}+Threshold)}$ where the thresholds is configured as 60 seconds. We follow this approach because it has been the value commonly used in the other research works. Like the *SLD*, it has no units.

4. The **Backfilled** variable holds if a given job has been backfilled or not when it was moved to the running queue. It holds *1* in affirmative case, *0* otherwise.

5. The **Nominal Runtime** variable holds the runtime of the job specified in the input workload. Its units are in *Seconds*.

6. The **Penalized Runtime** variable holds the amount of time that the job was penalized due to the saturation of resources sharing. It contains the amount of additional time the job is running in the evaluated configuration compared with an execution of the job in configuration with resources with infinite capacity. Note that the final runtime of the job $\alpha$ is computed as follows:
   $RunTime_{\alpha} = NominalRunTime_{\alpha} + PenalizedRunTime_{\alpha}$. Its units are in *Seconds*.

7. The **Percentage of Penalized Runtime** variable holds the percentage of the final runtime that corresponds to the penalized runtime. This variable has no units.

8. The **Computational Cost** variable holds the monetary cost associated to the execution of the job in the resource where it runs. This value is computed depending on the cost of the resource per hour and the final runtime of the job. For more information about the economic model see the chapter 8. This variable has currency units associated. Thus, it can represent *euros*, *dolars* or any other currency, depending on the specification of the configuration file.

[1]Note that some metrics were used in all the experiments, such as the Bounded Slowdown, but others were only used in some evaluations, such as the computational cost.

9. The **Computational Cost Respected** variable holds if the required monetary cost associated to the job execution specified by the user has been respected. Thus, if it has been greater than the final *Computational Cost* associated to the job. It holds *1* if it has been respected, *0* otherwise.

In the simulation configuration file the researcher can also specify if system metrics have to be collected or not. These kind of metrics contains information about the current status of the system in punctual time stamps of the simulation. They are collected by the Policy Data Collectors. As it is explained in the Alvio chapter (see chapter 4), the researcher can also specify the interval between two collections of these kind of variables [2]. Note that at the end of the simulation the researcher will have a set of instances of these variables resulting from the different collections. The policy metrics are:

1. The **Backfilled jobs** variable holds the number of backfilled jobs that have been backfilled in the last interval of time.

2. The **Jobs in Queue** variable holds the number of jobs that are currently queued in the wait queue of the system.

3. The **Processors used** variable holds the number of processors that are currently used by the running jobs.

4. The **Work Left** variable holds the amount of job that is pending on the wait queue. This amount is computed by adding the different areas of the queued jobs. The area is computed by multiplying the job requested time by the job number of requested processors.

5. The **Submitted Jobs** variable holds the number of jobs that have been submitted during this interval.

6. The **Load** variable holds the current load of the system.

7. The **Killed Jobs** variable holds the number of jobs that have been killed during this interval.

8. The **Running Jobs** variable holds the number of jobs that are currently running in the system.

[2] By default the collection is done each hour of simulated time.

## 3.3 Statistical Estimators

We applied a set of statistical estimators to each of the performance metrics that have been described in the previous subsection. In one hand, in the case of the job metrics these estimators are computed using all the jobs of the workloads. On the other hand, in the case of the policy metrics, these estimators have been computed using all the information collected in the different intervals. The estimators used are:

1. The **Median** is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and picking the middle one.

2. The **Mean** is the sum of the observations divided by the number of observations. It is a biased a estimator that should be used together the median since this last one is more robust against the outliers of that data.

3. The **Variance** of a sample is one measure of statistical dispersion, averaging the squared distance of its possible values from the expected value. Whereas the mean is a way to describe the location of a distribution, the variance is a way to capture its scale or degree of being spread out. The unit of variance is the square of the unit of the original variable.

4. The **Standard Deviation** is a measure of the spread of the values of the population, and it is defined as the square root of the variance.

5. The **Percentiles** $1..N$ are the values of a variable below which a certain percent of observations fall. For instance the percentile $25_{th}$ is the value of the observed population where a 25% of the observations are under this threshold.

6. The $95_{th}$ **Percentile** cuts off the top 5% values of the observed population. Usually, we use this estimator to evaluate the goodness of the system performance. The main reason is due to it represents almost all the observations of the simulation an discards the highest values (in most of the cases outliers).

7. The **IQR** (Interquartile difference) is computed by the difference of the $75_{th}$ and the $25_{th}$ quartiles. It is a non biased estimator that is used a measure of dispersion of the values of the population. This is commonly used instead of the standard deviation.

8. The **Maximum** returns the maximum value of the variable in the population.

9. The **Minimum** returns the minimum value of the variable in the population.
10. The **Accumulate** returns the sum of all the elements of the population.
11. The **Count Literals** counts all the different elements of the population. For example this estimator is used for count the different times that each resource selection policy has been used for allocate the jobs.
12. The **Accumulate Literals** sums all the observations of a given performance metric grouped by the nominal variable that is specified. For example, this estimator could be used for calculate the accumulate number of minutes of the jobs that have been allocated for each of the different resource selection policies.

More detailed information for the different properties and formalization of the statistical estimators summarized in this section see (94)(93)(102).

## 3.4 Workloads

### 3.4.1 Acknowledgment

All the information, including figures and description, that is described in this chapter has been collected from the *Parallel Workloads Archive* (28). The usage of the figures and descriptions shown in this chapter has been approved by Feitelson.

### 3.4.2 The Workloads, The Standard Workload Format and the Workload Archive

All the evaluations works presented in this Thesis used a set of well known workloads that have been evaluated in almost all the more representative scheduling strategies works. The main goal of using these workloads was to use all the previous knowledge concerning its behavior and characteristics in our evaluations. On the other hand, using these well known workloads allows to other researchers to carry out the same simulations or studies that we did. Thereby, they are able to reproduce our experiments and to compare their studies with results using the same source.

The database of workloads logs more used and referenced by all the researchers in job scheduling strategies is the *Parallel Workload Archive Logs* (28). This page points to detailed workload logs collected from large scale parallel systems in production use in various places around the world. For more detailed descriptions or characterizations the reader should access to this *Workload Archive*.

| | |
|---|---|
| Version | Version number of the standard format the file uses. |
| Computer | Brand and model of computer |
| Installation | Location of installation and machine name |
| Conversion | Name and email of whoever converted the log to the standard format. |
| MaxJobs | Integer, total number of jobs in this workload file. |
| MaxRecords | Integer, total number of records in this workload file. |
| Preemption | ’No’ , ’Yes’, ’Double’ (means that jobs may be split) and ’TS’ means time slicing is used, but no details are available. |
| UnixStartTime | When the log starts, in Unix time (seconds since the epoch) |
| StartTime | When the log starts, in human readable form, in this standard format (as printed by the UNIX ’date’ utility). |
| EndTime | When the log ends (the last termination), formatted like StartTime |
| MaxNodes | Integer, number of nodes in the computer. List the number of nodes in different partitions in parentheses if applicable. |
| MaxProcs | Integer, number of processors in the computer. |
| MaxRuntime | Integer, in seconds. This is the maximum that the system allowed, and may be larger than any specific job’s runtime in the workload. |
| MaxMemory | Integer, in kilobytes. |
| AllowOveruse | Boolean. ’Yes’ if a job may use more than it requested for any resource, ’No’ if it can’t. |
| MaxQueues | Integer, number of queues used. |
| Queues | A verbal description of the system’s queues. |
| Queue | A description of a single queue. |
| MaxPartitions | Integer, number of partitions used. |
| Partitions | A verbal description of the system’s partitions, to explain the partition number field. |
| Partition | Description of a single partition. |

Table 3.1: The Standard Workload Header

The original logs provided in the site come in different formats. Information about the individual format for each log is given to the degree available in the log’s associated notes document. In addition to the original format, all logs are converted to the Standard Workload Format (SWF) (29)(16). The standard workload format was defined in order to ease the use of workload logs and models. With it, programs that analyze workloads or simulate system scheduling need only be able to parse a single format, and can be

| | |
|---|---|
| Job Number | a counter field, starting from 1. |
| Submit Time | in seconds. The earliest time the log refers to is zero. |
| Wait Time | in seconds. The difference between the job's submit time and the time at which it actually began to run. |
| Run Time | in seconds. The wall clock time the job was running (end time minus start time). |
| Number of Allocated Processors | an integer. In most cases this is also the number of processors the job uses. |
| Average CPU Time Used | both user and system, in seconds. |
| Used Memory | in kilobytes. This is again the average per processor. |
| Requested Number of Processors. | Number of processors requested by the user at submission time. |
| Requested Time | This can be either runtime or average CPU time per processor. |
| Requested Memory | kilobytes per processor. |
| Status | 1 if the job was completed, 0 if it failed, and 5 if canceled. |
| User ID | a natural number, between one and the number of different users. |
| Group ID | a natural number, between one and the number of different groups. |
| Executable (Application) Number | a natural number, between one and the number of different applications appearing in the workload. |
| Queue Number | a natural number, between one and the number of different queues in the system. |
| Partition Number | a natural number, between one and the number of different partitions in the systems. |

Table 3.2: The Standard Workload Fields

applied to multiple workloads. As it is described by Feitelson, they are portable and easy to parse, the same format is used for models and logs and the format is completely defined, with no scope for user extensibility. A SWF trace is composed by two different parts:

1. The header: it contains comments with the format ";Label: Value". These are

special header comments with a fixed format that are used to define global aspects of the workload. The table 3.1 describe the most representative header labels that can be specified in a SWF trace file.

2. The data fields: contains the information about the execution of each of the jobs contained in the Workload. The fields that must be specified in the trace are presented in 3.2. When a field is not available for the current workload or for the current job a -1 is specified.

The logs that we used in our experiments are the cleaned versions of SWF logs generated with the original log. As it is described in the *Archive*, the SWF generated with the original files contains in some situations unrepresentative data, such as significant automated administrative activity or large-scale flurries of activity by single users. In order to ease the use of these logs in performance evaluations, Feitelson et al. also provide a cleaned version of the logs, and recommend that this version be used. More information about how these flurries are detected can be found in (100)(99)(37).

In the rest of the chapter we will provide a description about the different workloads that were used in the experiments of this Thesis. For each center we provide:

- A general description of the characteristics of the workload detailing where (in which center), how it was collected (the duration of the log, which environment characteristics) and the scheduling characteristics of the center (scheduling policy, partitions etc).

- A graphical description indicating the arrival ratio of the jobs (the amount of jobs submitted per hour and per each day of the week), the amount of processors requested per week, the degree of parallelism of the submitted jobs (the fraction of jobs that were executed in each number of processors), the size of the jobs that were submitted to the system (the cumulative distribution function associated to the runtime) and, finally, a scattered plot of the job size and job runtime.

### 3.4.3 The Los Alamos National Lab (LANL) CM-5 log

This log contains two years worth of accounting records produced by the DJM software running on the 1024-node CM-5 at Los Alamos National Lab (LANL). The total amount of jobs contained in the log is 79,302. It contains detailed information about resource requests and use, including memory. It also contains data on the user, executable, project, and submit, start, and end times. Jobs on the CM-5 use powers of two nodes according to a fixed partitioning. Gang scheduling is used, especially on smaller partitions, but jobs can also run in dedicated mode. The system is a 1024-node Connection Machine CM-5 system. Scheduling was performed by the DJM software.

The characterization of the workload:

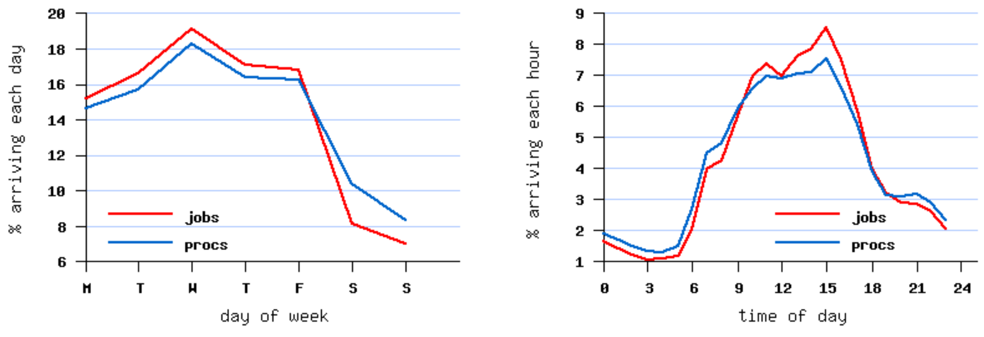
1. Figures 3.1a and 3.1b presents the arrival per day and per hour.

2. Figure 3.2 presents the runtime and level of parallelism description.

3. Figure 3.3 presents a scattered plot of the parallelism level against the runtime.

The jobs included in this workload require a high number of processors. As can be observed in the figure 3.2 the major number of jobs require 32 number of processors (50% of the whole workload) and the others requires up to 512. However, in general, these jobs are characterized by short runtimes (see figure 3.3), from one minute until 10 minutes. Note that only a 40% of the jobs have a runtime higher than 10 minutes. Concerning the arrival times of the jobs, from Monday until Friday around 20 jobs are submitted per day, concentrated in a interval of time of 7 hours (from 9 until 16).

We usually used this workload in scenarios where we wanted to evaluate the effect of workloads with high degree of parallelism. In some studies, to increase the fragmentation and demand of the system, we reduced the arrival times of the jobs until reaching a desired load.



(a) Arriving jobs per day (b) Arriving jobs per time

Figure 3.1: Arrival Ratio in the LALN workload

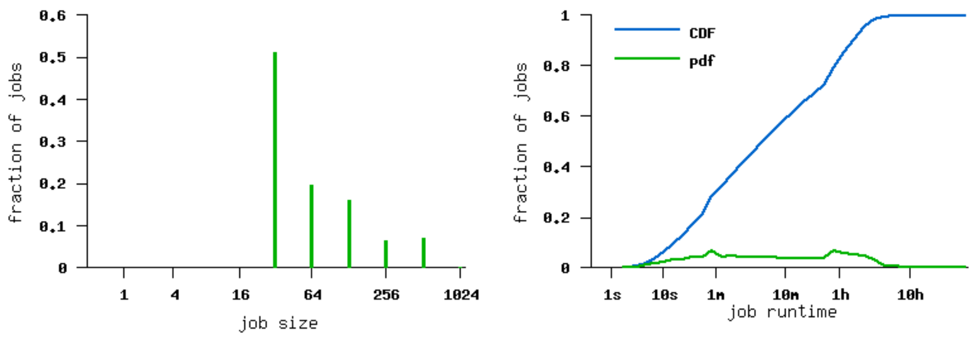


Figure 3.2: Job size and runtime in the LALN workload

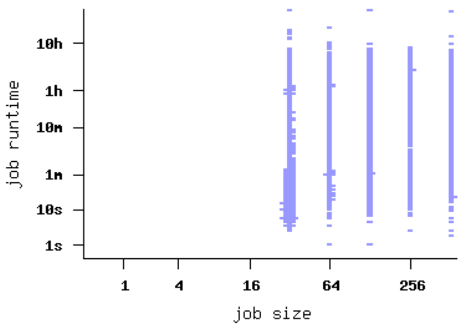


Figure 3.3: Scattered plot of runtime and job size in the LALN workload

### 3.4.4 The Cornell Theory Center (CTC) IBM SP2 log

This log contains 11 months worth of accounting records for the 512-node IBM SP2 located at the Cornell Theory Center (CTC). Scheduling on this machine was performed by EASY and LoadLeveler. The total number of jobs included in the log is 79300. Of the 512 nodes in the system, 430 are dedicated to running batch jobs. The remainder of the nodes are used for interactive jobs, I/O nodes, special projecs, and system testing. The log pertains to the batch partition.

The characterization of the workload:

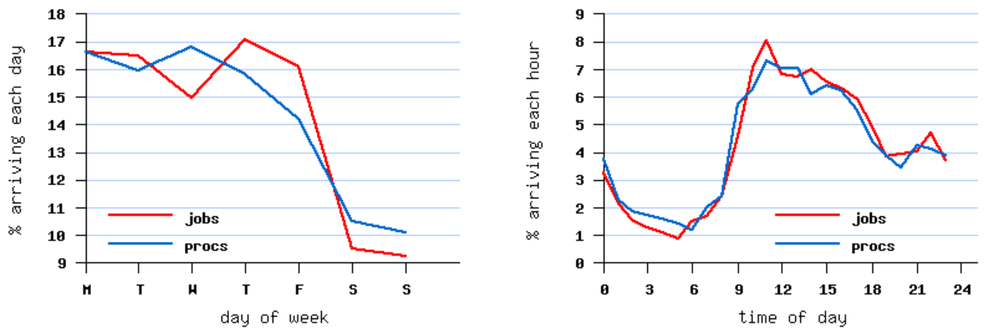
1. Figures 3.4a and 3.4b presents the arrival per day and per hour.

2. Figure 3.5 presents the runtime and level of parallelism description.

3. Figure 3.6 presents a scattered plot of the parallelism level against the runtime.

This workload, different from the previous one contains mainly a stream of sequential jobs with large runtimes. As can be observed in the figure 3.5 a 40% of the jobs are sequential jobs and the rest of the jobs require from 2 processors until 64 processors. Thereby, considering that the number of processors of the machine is 512, the degree of parallelism of its jobs is substantially lower than the LALN workload. On the other hand, the runtime of its jobs is substantially larger. The figure 3.6 shows how an important amount of the submitted jobs (mainly sequential jobs) have a runtime of more than ten hours. On the other hand a 40% of the jobs have a runtime higher than 1 hour and a 20% of them have a runtime higher than 10 hours (see figure 3.5) while in the LALN workload only the 20% of the jobs have a runtime larger than 1 hour, and only less than the 4% of the jobs have a runtime higher than 10 hours. The arrival time patterns of the jobs shown in this workload are very similar to the once presented in the LALN workload.

We used this workload to evaluate the impact of the stream of large jobs with relatively small degree of parallelism in the scheduling strategies.



(a) Arriving jobs per day (b) Arriving jobs per time

Figure 3.4: Arrival Ratio in the CTC workload

### 3.4.5 The Swedish Royal Institute of Technology (KTH) IBM SP2 log

This log contains eleven months worth of accounting records from the 100-node IBM SP2 at the Swedish Royal Institute of Technology (KTH) in Stockholm. The total amount of jobs included in the log is 28940. The system imposes limits on job run times,

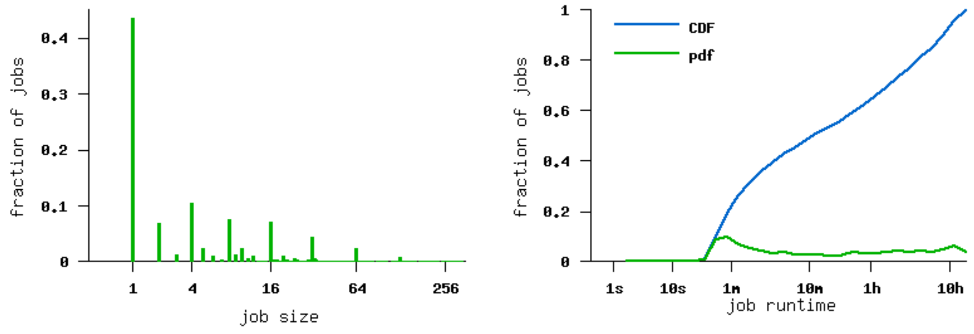


Figure 3.5: Job size and runtime in the CTC workload

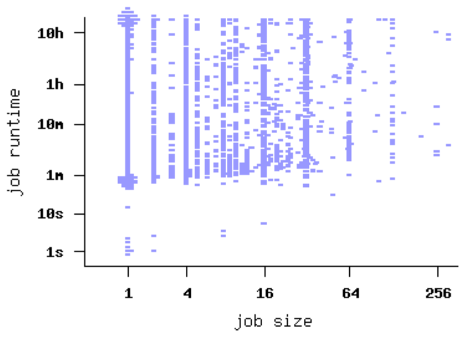


Figure 3.6: Scattered plot of runtime and job size in the CTC workload

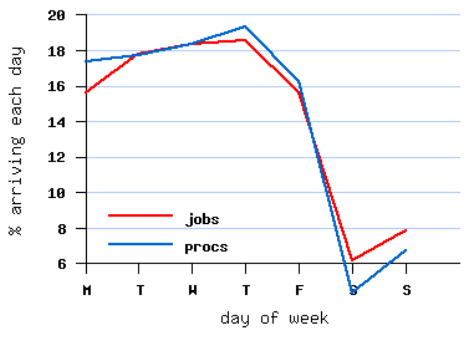
and this was changes a couple of times during the period that the log was recorded. For more information about these limits visit the *Workload Archive*.

The characterization of the workload:

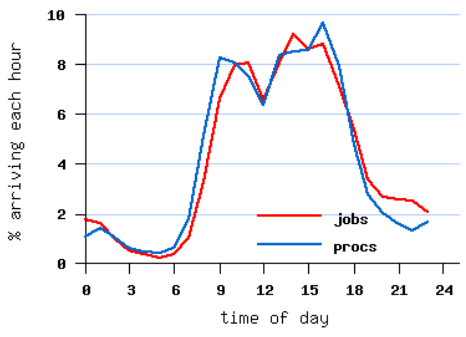
1. Figures 3.7a and 3.7b presents the arrival per day and per hour.
2. Figure 3.8 presents the runtime and level of parallelism description.
3. Figure 3.9 presents a scattered plot of the parallelism level against the runtime.

The degree of parallelism of the jobs included in this workload is more diversified. However, still an important amount of the submitted jobs is sequential (around the 30%)

or with low number of processors (i.e: a 10% of the jobs only require two processors). The runtimes shown in this workload are also relatively small. The 40% of the jobs has a runtime less than 1 minute. As can observed in the figure 3.9, this workload shows more diversity in terms of degree of parallelism and runtime of the submitted jobs. Note that the rest of the scattered plots of the other two workloads present a set of clouds showing that there are several combinations of processor/runtime that are more frequent in the system. However, this workload shows all the values spread among the entire plot. We used this workload to evaluate the impact of having a workload with more diverse typologies of jobs.

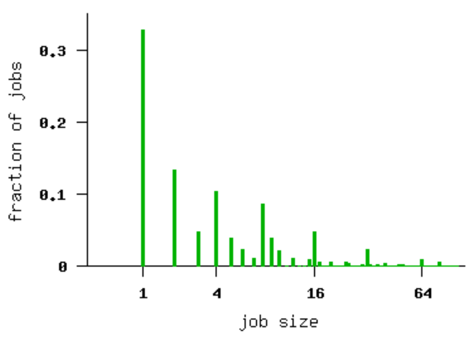


(a) Arriving jobs per day



(b) Arriving jobs per time

Figure 3.7: Arrival Ratio in the KTH workload



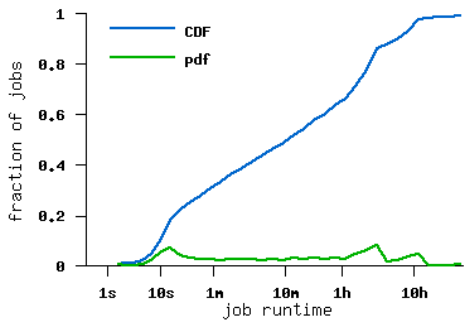


Figure 3.8: Job size and runtime in the KTH workload

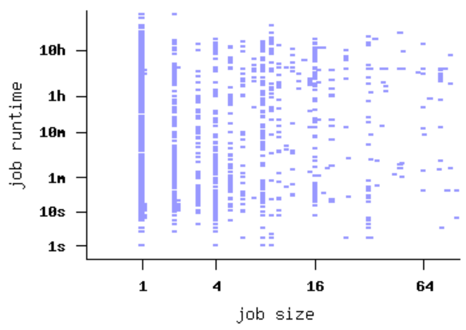


Figure 3.9: Scattered plot of runtime and job size in the KTH workload

### 3.4.6 The San Diego Supercomputer Center (SDSC) Blue Horizon log

An extensive log, starting when the machine was just installed, and then covering more than two years of production use. The total amount of jobs included in the log is 250400. It contains information on the requested and used nodes and time, CPU time, submit, wait and run times, and user. The total machine size is 144 nodes. Each is an 8-way SMP with a crossbar connecting the processors to a shared memory. These nodes are for batch use, with jobs submitted using LoadLeveler. The data available here comes from LoadLeveler. The scheduler used on the machine is called Catalina. This was developed at SDSC, and is similar to other batch schedulers. It uses a priority queue, performs backfilling, and supports reservations.

The characterization of the workload:

1. Figures 3.10a and 3.10b presents the arrival per day and per hour.

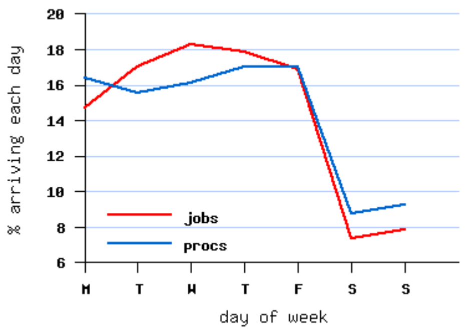
2. Figure 3.11 presents the runtime and level of parallelism description.

3. Figure 3.12 presents a scattered plot of the parallelism level against the runtime.

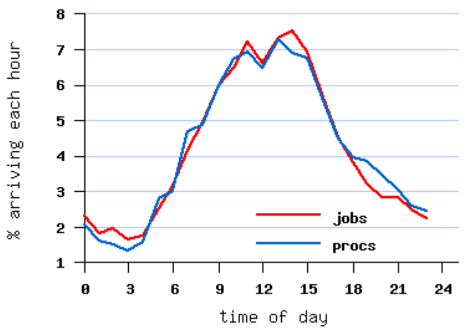
The jobs included in this workload certainly have a substantially larger and diversified runtimes that the jobs of the two previous workloads. It contains runtimes from few seconds until jobs of four days. As can be observed in the figure 3.11 the number of sequential jobs is almost null. Mainly the jobs that are submitted to this system use 8 processors (around the 50% of the jobs). The major part of these jobs is composed by short jobs that run for one minute. The runtime of the rest of the jobs is approximately larger than ten minutes. In all the different degrees of parallelism there are a representative number of jobs that run for more than ten hours (see the top part of the figure 3.12). Comparing this scattered plot with the once of the previous workloads, this workload clearly shows jobs with more HPC demand (in terms of runtime and processors).

Finally, the arriving jobs per time and day (see figures 3.10a and 3.10b) show similar patterns to the previous workloads. However, different from the other workloads, here the users of the system show submission activity until 21.

We used this workload for evaluate the impact of having stream of non sequential jobs with relatively large runtime. Note that this workload holds jobs that can potentially show more impact in the performance of the scheduling policy.



(a) Arriving jobs per day



(b) Arriving jobs per time

Figure 3.10: Arrival Ratio in the SDSC-BLUE workload

### 3.4.7 The San-Diego Supercomputer Center (SDSC) Paragon

This log contains two years worth of accounting records for the 416-node Intel Paragon located at the San Diego Supercomputer Center (SDSC). The total amount of jobs contained in the workload is 76872 in 1995 and 38723 in 1996. Due to historical reasons, the log is divided to two parts (one per year). These extensive logs contain information about the number of nodes, submit, start, and end times, CPU time used, NQS queues used and their limits, and user. There is no information about the application being run. The Paragon is a mesh with processing nodes based on the Intel i860 processor. The SDSC system has 416 nodes.

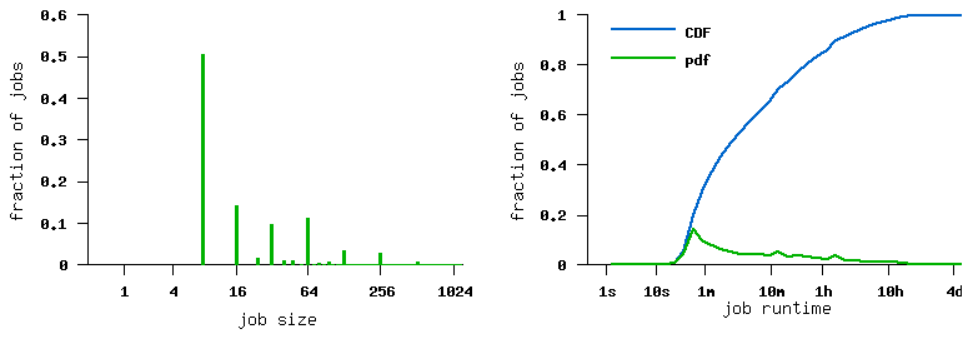


Figure 3.11: Job size and runtime in the SDSC-BLUE workload

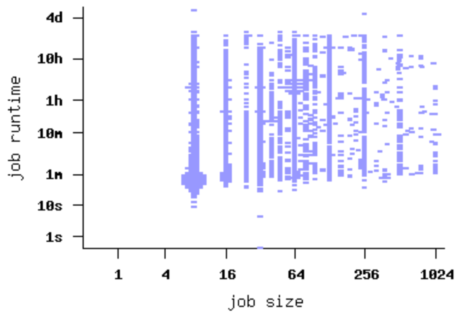


Figure 3.12: Scattered plot of runtime and job size in the SDSC-BLUE workload

The characterization of the workload for the year 1995:

1. Figures 3.13a and 3.13b presents the arrival per day and per hour.
2. Figure 3.14 presents the runtime and level of parallelism description.
3. Figure 3.15 presents a scattered plot of the parallelism level against the runtime.

The characterization of the workload for the year 1996:

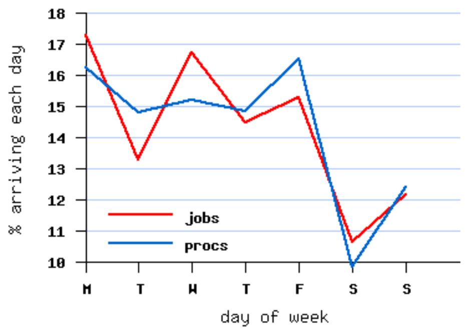
1. Figures 3.16a and 3.16b presents the arrival per day and per hour.
2. Figure 3.17 presents the runtime and level of parallelism description.

3. Figure 3.18 presents a scattered plot of the parallelism level against the runtime.

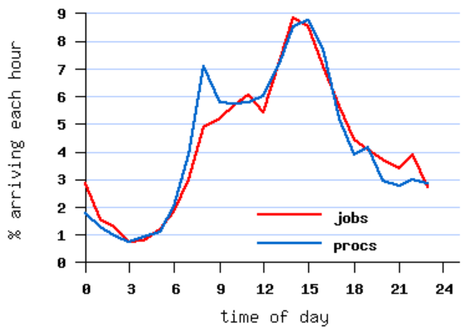
Jobs of both workloads do not show a clear tendency in the usage of determined degree of parallelism. However, the workload of the year 95 shows a major tendency on the submission of sequential jobs. In both cases, the numbers of processors showing higher fractions of jobs are 4, 8 and 16. Concerning the runtime, the first year shows similar pattern that the KTH workload (80% of the jobs have a runtime lower than one hour). However, the second year shows a relevant increase in runtime for the submitted jobs. Around the 40% of the jobs have runtime higher than one hour, while in the previous year this fraction is about the 10%. The scattered plot of the first year (see figure 3.15) shows how the size of the submitted jobs and the required number of processor are relatively low. The scattered plot presents an accumulation of points in the lower and left part of the plot. On the other hand, the scattered plot of the year 96 (see figure 3.18) of the workload shows more diversified typology of jobs.

Concerning the arriving jobs per day and per time shows similar patterns in both years. However, compared to the other workloads it show a slightly lower number of submitted jobs per hour and day.

Using the workload containing both years of execution, we evaluated how a given scheduling strategy is able to adapt the change in the characteristics of the streams of jobs that are submitted to the system.



(a) Arriving jobs per day



(b) Arriving jobs per time

Figure 3.13: Arrival Ratio in the SDSC95 workload

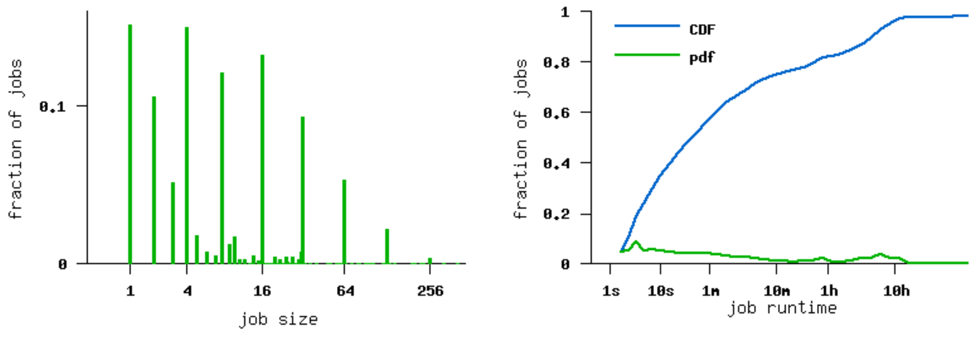


Figure 3.14: Job size and runtime in the SDSC95 workload

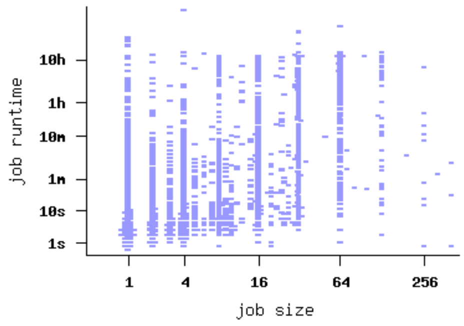
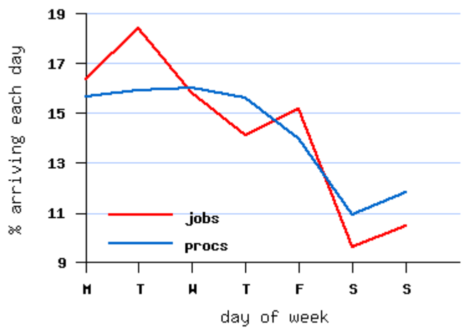
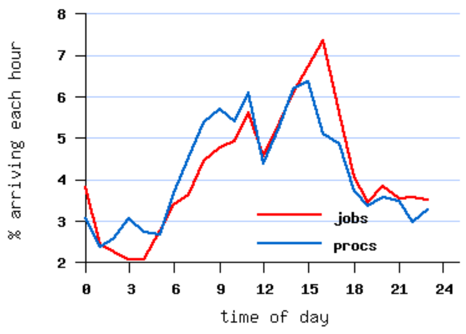


Figure 3.15: Scattered plot of runtime and job size in the SDSC95 workload



(a) Arriving jobs per day



(b) Arriving jobs per time

Figure 3.16: Arrival Ratio in the SDSC96 workload

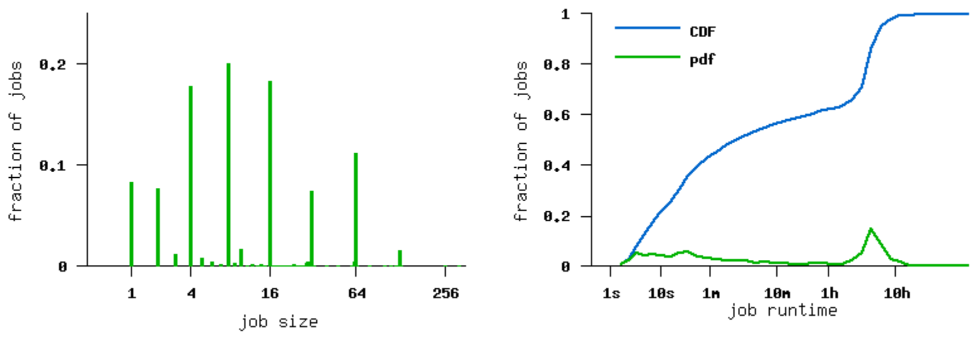


Figure 3.17: Job size and runtime in the SDSC96 workload

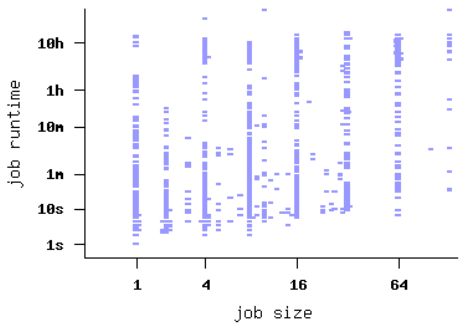


Figure 3.18: Scattered plot of runtime and job size in the SDSC96 workload

Chapter 4

# The Alvio Simulator

The Alvio Simulator is a C++ event driven simulator that has been designed and developed to evaluate Scheduling Policies in High Performance Architectures. Like other simulators, given a workload and an architecture definition, it is able to simulate how the jobs would be scheduled using a specific scheduling policy (such as First-Come-First-Serve, or the Backfilling policies). The main contribution of this simulator is that it not only allows modelling the job workflow in the system, it also allows simulating different resource allocation policies (such as First-Continuous-Fit). As a result, the researcher is able to validate how different combinations of scheduling policies and resource selection policies impact on the performance of the system. The other new contribution of this simulator is the modelization of resource usage of the architecture by different jobs that are running in the system. The researcher is able to specify the resources that are available in the architecture (for instance the amount of memory bandwidth in each node) and the amount of resources to be used for each job in the workload. In this chapter we describe the internal structure of the simulator, its functionalities and the different models that are currently available to researchers to evaluate different High Performance Architecture scenarios.

## 4.1 The Alvio structure

The Alvio simulator is divided into six different modules that are conceptually related to six different semantic domains in the simulations: statistical analysis, prediction techniques, data mining techniques, architecture modelization, simulation modules, and finally, general utilities. The modules are:

- The *Statistical Module* provides a set of functionalities that allows statistical methods to be applied to sets of data. The module provides a group of predefined statistical estimators that can be used by any of the other simulator components at any part of the program execution. These techniques are mainly applied to the output simulation variables. Furthermore, the simulator can carry out more complex analysis on the variables using the capabilities provided by the R Statistical Package (65)(24). We provide the researcher with a set of predefined analysis that can be used for more complex analysis of this output data, for example to generate a set of graphics which provide a time analysis of the evolution of the system during the whole simulation.

- The *Data Mining Module* provides a set of components that allows data mining algorithms to be used on sets of data. For example, some classification techniques (such as the ID3 trees or the C45 trees) are available. The techniques that are currently provided are implemented on top of the Weka (63). This module also provides a set of generic interfaces to include new models or techniques based on the Weka classes, and therefore, extending the current model does not imply an important effort in terms of development and design.

- The *Prediction Module* provides a set of functionalities that allows prediction techniques to be used on a set of variables based on historical information. For example, some of the scheduling policies that are provided in the simulator can use the predictor service to estimate job runtime and to decide how to schedule the job. The approach that we have followed in the design of the predictor internals, how the scheduler manages missed deadlines, and how the predictor and the scheduler interact, is based on the formalizations provided in the Tsafrir paper (98).

- The *Utile Module* provides a set of generic utilities that can be used by different simulator components. Almost all of them are designed to manage file streams, load, modify and validate XML documents, load, modify and save CSV documents, etc.. This module also provides specific components that are built on top of the generic functionalities. For example, a paraver trace generator, a logging module or the CSV job performance file generator.

- The *Architecture Module* provides all the necessary abstractions necessary to model different high performance architectures where jobs are allocated in the simulations. Using these abstractions the simulator can provide the researcher with a set of particular architecture definitions. As will be introduced later, the researcher is able to specify, for example, the memory bandwidth of all the nodes of the simulated system. Currently, one model is provided: the cluster architecture, this is described later in more detail.

- The *Simulation Module* is the core module of the simulator and provides the necessary functionalities to instantiate simulations. It contains the main components related to the simulation techniques, such as the event definition, the event processing, the simulation pre-processing and post processing. Furthermore, it contains the modelization of all the available job scheduling policies and resource selection policies.

In the rest of this section we provide a more detailed description of each of the modules introduced above.

## 4.2 The Statistical Module

As explained above, the Statistical Module provides a group of estimators and a set of complex analysis that can be applied to a given variable. In this subsection we provide a more detailed description of all the statistical techniques that the simulator offers researchers wishing to analyse simulation results.

### 4.2.1 The data collectors

In the simulation configuration file the researcher can specify which variables are collected during the simulation. The simulation variables that can be analyzed when the simulation finishes are grouped into two different kinds of entities: the Job entity and the Policy entity. The first group of variables contains the performance that all the jobs have experienced in the simulation. The second group of variables contains information about the status of the system (or systems in the case that the simulated architecture is a meta-scheduling scenario) during the simulation. The variables available to each of these two entities are:

- The Job entity has these variables: bounded slowdown, slowdown, wait time, penalized runtime, percentage of penalized runtime, resource selection policy, if the job has been backfilled, if the job has been killed, if the job has been completed and the center to which the job has been submitted.

- The Policy Center entity has these variables: the number of jobs in queue, the number of backfilled jobs, the amount of pending work (where it is computed as $PW = \sum_{\forall jobs} RequestedTime_{job} * Processor_{job}$) and the name of the center simulated.

The statistical module contains the *Collectors Sub module*. The Collectors are responsible for collecting information from the different entities during the simulation and storing the information in CSV files. In the current architecture two different collectors are defined. The *Job Info* collector collects job performance information during the simulation. When a job finishes, the collector gathers the performance information and adds it to the simulator data base that will afterwards be dumped into the CSV files and analyzed by the main component of the module. The *Policy Info* collector collects the system performance information during the simulation. In the simulation configuration file, the researcher can specify the interval of time between two consecutive collections. By default, the system performance variables are collected hourly, but for long simulations the use of a longer interval is recommended.

### 4.2.2 The estimators

The simulator provides a set of statistical estimators that can be applied to all the output simulation variables, for example, to job slowdown. For each of these variables, the researcher can specify in the configuration file which estimators have to be applied. For example, he/she could specify that the wait time for all the jobs has to be collected, and that the average and median statistical estimators have to be applied to this variable. Given the simulation configuration file, the core component of this sub module will apply the desired specified statistical analysis. It is important to emphasize that not all the estimators can be applied to all the performance variables. For example the mean estimator cannot be applied to the variable *submitted center* for the entity job, since this is a nominal variable.

Depending on the type of variable that has to be analyzed (i.e.: the variable slowdown of the entity performance is a continuous variable) the researcher can use any of the following estimators:

- The **Accumulate Literals** estimator, given a vector of pairs $\{<string, double>\}$, returns the addition of the doubles grouped by second the strings of the pair. For example, given the vector $\{<2, CTC>; <1, SDSC>; <4, CTC>\}$ the estimator would return $\{<6, CTC>; <1, SDSC>\}$.

- The **Average** estimator, given a vector of real values, returns their average. For example, given the vector $\{<1, 4, 4>\}$ would return the value $\{<3>\}$.

- The **Standard Deviation** estimator, given a vector of real values, will return their standard deviation. For example, given the vector of the previous example the estimator would return $1,73$.

- The **Count Literals** estimator, given a vector of string, counts the number of times that each of the literals appears in the container. For example, given the vector $\{CTC;SDSC;CTC\}$ the estimator would return $\{<2,CTC>;<1,SDSC>\}$.

- The **Percentiles** estimator, given a vector of real values, returns a vector containing the percentiles from the $10_{th}$ until the $90_{th}$ incrementing in 10 each percentile. A percentile is the value of a variable below which a certain percentage of observations fall. For example, the 20th percentile is the value (or score) below which 20 percent of the observations may be found. For example, given the vector $\{<1,2,67,90,90>\}$ the estimator would return $\{<1,1,2,2,67,67,90,90,90>\}$.

- The **Percentile 95** estimator, given a vector of real values, returns the value containing the percentile $95_{th}$ of all the values included in the container. This estimator is the most commonly used in the simulation evaluation as it is able to represent most of the population included in the input data and it is not affected by the outliers.

- The **Median** estimator, given a set of real values, will return the $50_{th}$ of all the values included in the container. This estimator is the non-biased estimator of the average or mean. In the vector described in the previous example, the median is 67.

- The **Interquartile difference** estimator, given a vector of real values, returns the difference between the $percentile_{75}$ and the $percentile_{25}$. For example, given the vector $\{<1,1,2,2>\}$ the estimator would return 1. This estimator is the non-biased estimator of the standard deviation.

- The **Max** estimator, given a vector of real values, returns the higher value included in the container.

- The **Min** estimator, given a vector of real values, returns the lower value included in the container.

- The **Accumulate** estimator, given a vector ore real values, returns the adding of all the values included in the container. For example, given the vector $\{<1,10,2,2,0.2>\}$, the estimator would return $15,2$.

We have defined a generic interface which specifies how an estimator has to be designed, developed and included in the simulator. Using this interface, the researcher can easily develop his/her own statistical estimators: all components of the simulator are able to use the new estimators using this generic interface. The methods that this interface defines are:

```
virtual Metric computeValue ()=0;
Simulator::metric_t getmetricType () const;
Simulator::statistic_t getstatisticType () const;
void setMetricType (Simulator::metric_t &theValue);
void setStatisticType (Simulator::statistic_t &theValue);
```

The first method computes the estimator which a specific estimator implements. This method returns a metric that contains the result of the computation. This has to be called when the estimator has all the necessary data to compute the algorithm. The way the estimator retrieves the data depends on the specific implementation of the estimator. For example, the Max estimator has the two set methods detailed below. These give the estimator the values which the average has to be computed from. Given an instantiation of a specific estimator, the components of the system know the type of metric that is returned by the algorithm (i.e.: double, string or real) and the statistic that is implemented by the current instantiation (i.e.: AVG, STDEV or IQR).

```
virtual Simulator::Metric computeValue ();
vector< double > getdvalues () const;
vector< int >  getivalues () const;
vector< double >  getvalues () const;
void setDvalues (vector< double > theValue);
void setIvalues (vector< int > theValue);
void setValues (vector< int > theValue);
void setValues (vector< double > theValue);
```

Figure 4.1 provides a UML based description of the different estimators that are available in the simulator and how they extend the basic interface that is specified in the generic estimator.

### 4.2.3 The R Analyzers

The statistical estimators that are provided in the module can provide some information about how some variables, such as job wait time, behaved during the simulation. Although the researcher can derive some information using the percentiles or the interquantile difference, our experience shows that in some situations this data is

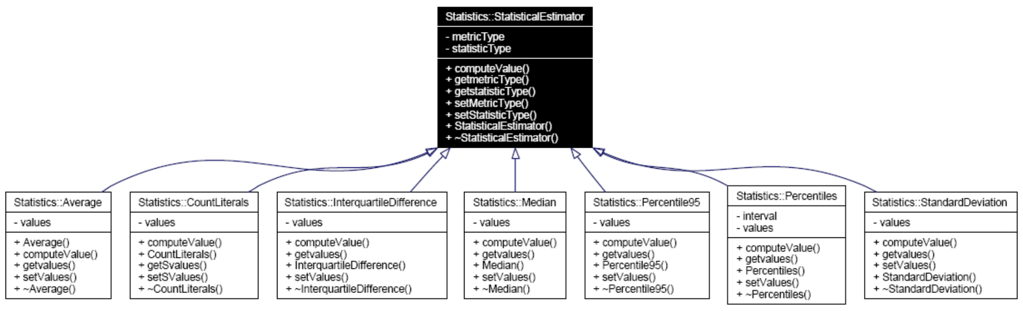


Figure 4.1: The statistical estimators

not enough to understand some *mysteries* that can occur in determined points of the simulation.

We have included the possibility of using more complex analysis of the output simulation data. This analysis is based on a set of predefined R scripts which have been designed for specific analysis of CSV fields with a certain structure and information. The process of analyzing the variables contained in the CSV file (such as job slowdown) generates plain text files and PDF files containing a graphical and statistical analysis of the whole simulation. Using this derived information, the researcher is able to study in more detail the evolution of these variables during simulation time. For example, when the analyzed file contains the performance achieved for each queue during the simulation, the job extended analyzer provides a graph showing the evolution of the jobs in queue per hour during the simulation, see figure 4.2. Observing this figure the researcher can detect one interval of the simulation where the number of queued jobs has increased suspiciously. Thus he/she can focus his/her analysis on the jobs that were scheduled in this interval of time.

The simulator provides three different kinds of analysis:

- The job performance analysis provides a summarized analysis of the most representative variables of the CSV file containing information of all jobs that have been scheduled in the system. It includes:
  - The evolution during the simulation of variables slowdown, bounded slowdown, job wait time and penalized runtime (see below).
  - An histogram of the variable job simulation finish status. It indicates the number of jobs that have been killed, completed, cancelled etc.
  - An histogram of the variable submission center. If the simulation models

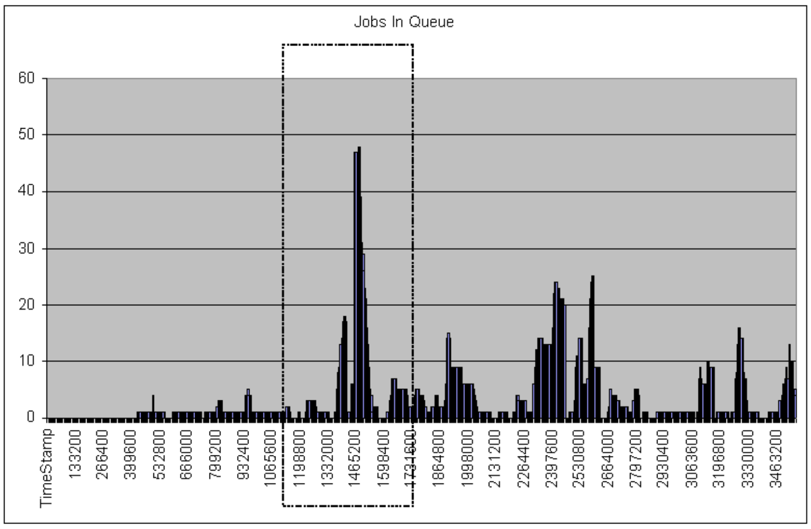


Figure 4.2: A sample of the R Analyzer output

a distributed environment it will show the number of jobs that will be submitted to each center.

– An automatic clustering analysis about the wait times for the queued jobs. This analysis can help to detect tendencies in the system. Thereby finding different streams of jobs or circumstances that may affect the performance.

- The local system performance analysis provides an analysis of the most representative variables of the CSV file containing information of the system during the simulation. It includes:
  – The evolution during the simulation of the variable number of jobs in the number of queued jobs in each queue of the center, the number of processors used, the number of backfilled jobs, the amount of pending work and the number of queued jobs.
- The distributed system performance provides the previous analysis for each of the centers available in a distributed architecture.

Figure 4.3 gives a summary of the internal structure of this R sub module. It has two main components:

- The CSV Processor is the component that loads the CSV files required to be analyzed. Generally, they are the output CSV simulation files. Thus, the CSV files with information about job performance and the CSV file containing information about the system during the simulation are loaded.
- The Analyzers are the components responsible for applying a given R Script to the specified CVS file.
- The Core *R*-Processor Engine (CRPE) is the main component of the sub module. Once the simulation has finished, and given the simulation configuration, it will load all the CSV output files, will instantiate the required R Analyzers, and will orchestrate all the analysis.

Given the configuration of the simulation and using the necessary CSV Processor, the CRPE will load and validate the necessary CSV files and will process them based on the specified analysis using the appropriate Analyzers. For example, if the *JOB_CSV_R_ANALYZER_BASIC* analyzer is specified in the simulation configuration, the CSV Processor that processes the CSV file with the jobs performance will be loaded by the CRPE. The CSV file will be provided to the Processor. And finally, the Processor will be pipelined for the Analyzer to the appropriate R Scripts (in the order specified by the Analyzer in the Simulator configuration file).

To add new R Scripts Analyzers to the simulator the researcher has only to edit the Simulator configuration file indicating the name of the new Analyzer, the file path of the R script and the CSV file types that are required by the analyzer (see the following example). The name of the Analyzer will be used later in the simulation configuration file.

```
<RPostProcessingFiles>
  ...
  <Script>
    <AnalyzerType>ANALYZER_NAME</AnalyzerType>
      <ScriptPath> \%FILE_PAHT\Script.r</ScriptPath>
      <CSVFileType>CSV_FILE_TYPE</CSVFileType>
  </Script>
</RPostProcessingFiles>
```

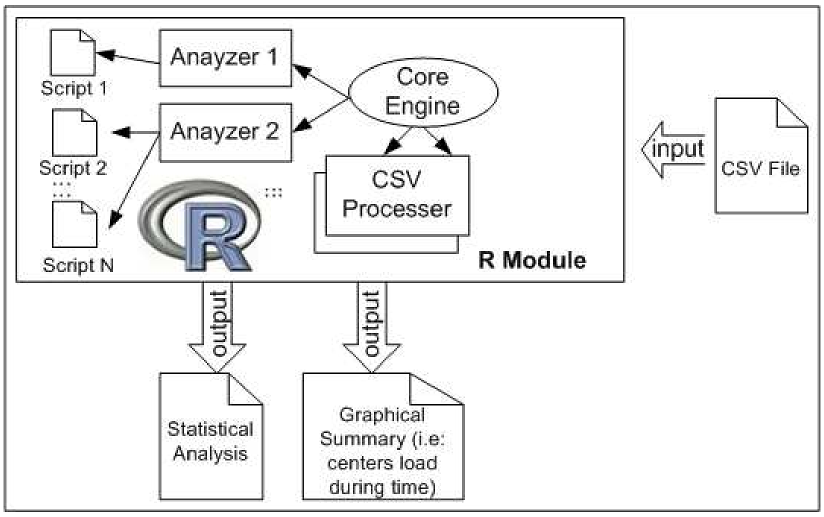


Figure 4.3: The R Analyzer sub module

## 4.3 The Data Mining Module

This module provides a set of components that apply data mining techniques to a set of data to derive information. The current architecture provides functionalities to build classification models and automatic discretization methods for continuous variables. These components are implemented following an abstract interface that has been designed for each technique. Thus, the researcher can implement new discretization or classification components following these interfaces.

#### 4.3.0.1 The abstract interfaces

The next source shows the abstract interfaces that the researcher should implement when defining a new descretizer component. Two major kinds of methods must be implemented: the first group of methods (lines 1-3) allow the researcher to gather and set the information according to its properties (such as its name, or the type of discretizer); while the second group (line 6) provide the functionality itself. Given an enumeration of continuous values, the method will return a vector with the nominal values resulting from the applied discretization algorithm.

```
1 Discretizer(Log* log);
2 ~Discretizer();
3 void setDiscretizerType(const discretizer_t& theValue);
```

```
discretizer_t getdiscretizerType() const;
string getDiscretizerName(discretizer_t classifier);
virtual vector<string>* discretize(vector<double>* in)=0;
```

The next source shows the abstract interfaces which the researcher should implement when defining a new classifier component. The classifier interface is more complex than the previous once. Basically, this is because almost all the classifiers use historical information to carry out the classification method. As can be observed in line 1, the constructor for this abstract class has more information than the discretizer constructor:

- The first parameter is of a vector containing the names for the different attributes that will be used to build the classification model. These attributes will be used as the response variables.

- The second parameter is a vector that indicates the types of each of the preceding attributes. The classification method will automatically discretize all the variables that are continuous to nominal variables.

- The fourth parameter indicates the type of discretizer that has to be used to convert the continuous variables to nominal variables. In the case that the default is specified, the continuous variables will be discretized using intervals with the same size.

- The last parameter indicates the amount of bins that have to be used when discretizing the continuous variables. [1]

The remaining methods are designed to gather the information about the classification method provided by the component (lines 3-5), to build the model (line 6-7), and to query the model for a classification (line 8). Usually, the classification methods require a minimum amount of historical information to build the model, thus, each time a classification is known this information has to be provided to the component using the method shown in line 6. For example, in a classifier instantiation where the response variables are *number of processors, the executable name, the amount of memory used and the user name* of the job and the prediction variable is its *runtime*, the component could report that the job with 12 processors, executable *BLAST*, *[12,32)* MB of memory used for the user *fguim* has a runtime *[12,43)* minutes. When the component has enough information to create the model, the function *generateModel* can be called. Usually, as the amount of historical data available during the time increases, this last method is

[1]For more information concerning discretization methods see (63)

required periodically. As a result, the quality of the classification models is expected to increase over time.

```
Classifier(vector<string>* attributes,
           vector<native_t>* attributesTypes,
          Log* log, discretizer_t discretizer, int numBins);
~Classifier();
void setClassifierType(const classifier_t& theValue);
classifier_t getclassifierType() const;
string getClassifierName(classifier_t classifier);
virtual bool addInstance(vector<string>* Instances)=0;
virtual bool generateModel()=0;
virtual string classify(vector<string>* RespAttributes)=0;
```

In the presented version of the simulator, the data mining techniques that are included are the two presented in this sub subsection. However, the researcher can also add new typologies (i.e.: defining a generic interface for use with clustering models) of components in the simulator following the same philosophy as that of the presented abstract interfaces.

The main advantage of using generic interfaces to work with different data mining functionalities is that the researcher is able to use different techniques, probably with different insights using a common interface. For example, as is described in subsection *4.3.2* the simulator provides two types of classifiers with different properties and different classification algorithms. However, when he/she has to interact with both kinds of classifiers they follow the same interface. As a result, the only choice that has to be taken is which classifier should be used.

### 4.3.1 Integrating the Weka Framework

We have already mentioned that the simulator provides implementations of both types of components which inherit the definitions described in the previous subsection. These components were developed using the Weka data mining framework. As is described in the Weka website, Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a data set or called from one's own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes.

In general, the data sets that are used by the Weka tools (classifiers, cluster modules etc.) are provided by the simulator using the *ARFF* files. These files, similar to the traditional CSV files, provide row based information, where each row contains the information of a given instance. However, in contrast to CSV files, it contains a detailed description of the native type of each of the instances attributes.

To allow the generic use of the Weka components inside the simulation sources, we created a set of functionalities and bindings that allow access to the different functionalities provided by the different Weka *java* classes. The main component of the simulator which provides transparent access to the Weka and allows use of its java classes is the *WekaObject*. The source presented above shows the most important functionalities that this binder provides. Three types of functions are available: functions that set up the environment and use the Weka classes (lines 1-2); functions that call the Weka functionalities (lines 5-7) and functions that generate and manage the *ARFF* files (lines 8-24).

```
1  void setWekaEnv(string JavaBinary,
2                  string classpath,
3                  string wekaPath,
4                  string TempDir);
5  vector<string>* InvoqueWekaFunctionality(string WekaClass,
6                  string Parameters,
7                  bool neededOutput);
8  void readAttributesDeffinition(string file,
9                  vector<string>* output);
10 string getIntervalFromReal(string attributeDefinition,
11                 string real);
12 bool generateArffFile(string FilePath,
13                 InstancesSet* instances,
14                 vector<string>* attributes,
15                 vector<native_t>* attributesTypes,
16                 int numberOfInstances);
17 bool generateArffFileWithHeader(string FilePath,
18                 InstancesSet* instances,
19                 vector<string>* headerEntries,
20                 int numberOfInstances);
21 bool saveArffFile(string FilePath,vector<string>* AttLines,
22                 vector<string>* InstanceLines);
23 InstancesSet* loadInstancesArffFile(string FilePath,
24                 int NumberAttributes);
```

### 4.3.2 The available classifiers and discretizers

Using the interfaces and the *WekaObject* introduced in the previous two sections we built two different kind of classifiers and discretizer:

- The *EqualFrequency* discretizer discretizes the numeric attribute using the same number of instances per bin. It automatically finds the number of intervals in which the attributes will be discretized if required in the instantiation of the component. Otherwise the number of bins provided is used.
- The *EqualIntervalLengh* discretizer discretizes the numeric attribute using the same length for all the intervals. As with the previous case, it automatically finds the number of intervals in which the attributes will be discretized if required in the instantiation of the component. Otherwise the number of bins provided is used.
- The *ID3* and *C45* classifiers provide a classification component based on the ID3 and C45 algorithms respectively. More information about both algorithms can be found in (101)(25).

The internal design of both kinds of Weka components have a very similar architecture. Below we present a detailed description of how the classifiers are built on top of the Weka functionalities, their architecture, and the interaction between all the different elements. The discretizer insights are a simplified version of the presented ones.

Figures 4.4 and 4.5 present the different elements that compose the Weka classifiers. The other simulator components can interact with them using the previously described abstract interface. When the generation of the model is required (Figure 4.4) the Weka based classifier will dump the instances that have been previously provided by the external component, and, using the Weka Object functionalities, will generate the *ARFF* file containing their information. Once the ARFF file is generated, again using the Weka Object, the Classifier will call the appropriate Weka class providing the desired parameters and the file to generate the parameter (in case of the ID3 the *weka.jar weka.classifiers.trees.id3* will be used, and in case of the C4.5 trees the *weka.classifiers.trees.J48*). The output of the model construction is a "'.model"' binary file containing the classification tree generated. Once the model is generated, following the same approach, the Classifier can ask for a classification of a given instance, providing the model file to the Weka classes (Figure 4.5).

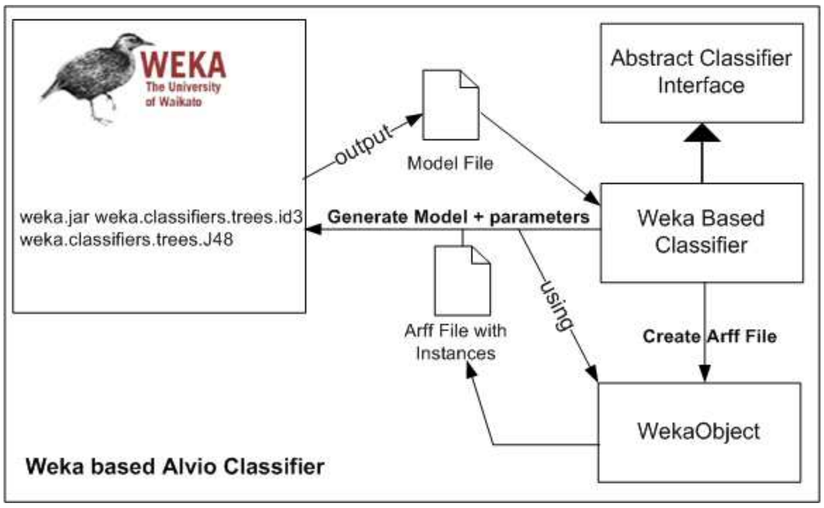

WEKA
The University of Waikato
weka.jar weka.classifiers.trees.id3
weka.classifiers.trees.J48
output
Model File
Generate Model + parameters
Arff File with Instances
using
Abstract Classifier Interface
Weka Based Classifier
Create Arff File
WekaObject
Weka based Alvio Classifier

Figure 4.4: Generating a Classification Model

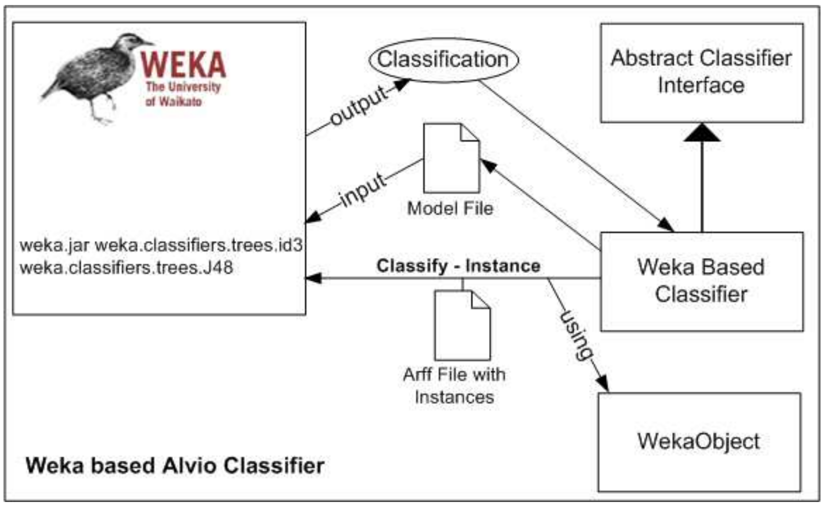

WEKA
The University of Waikato
weka.jar weka.classifiers.trees.id3
weka.classifiers.trees.J48
Classification
output
input
Model File
Classify - Instance
Arff File with Instances
using
Abstract Classifier Interface
Weka Based Classifier
WekaObject
Weka based Alvio Classifier

Figure 4.5: Using a Classification Model

## 4.4 Prediction Module

The prediction module has been designed to provide scheduling policies with the opportunity to receive more information about scheduled jobs prior to their execution. For example, on job submission time, schedulers may use a predicted runtime rather than the usually used estimated job runtime provided by the user. In this section we present the components that are included inside this module and explain how they can be used by the other components of the simulator.

### 4.4.1 The abstract interfaces

The philosophy that has been followed in the design of the predictors structure is based on the approach that Tsafrir and Feitelson proposed in (98). Figure 4.6 shows a generic view of the interaction between the scheduler and the predictor component. As can be observed, the interaction between both elements is based on three main functionalities:

- Given a characterization of a job, the predictor's *Predict* functionality will carry out an estimation of the required variable (in the explanations we will assume that the variable is the runtime, however other variables can be predicted). For example, the scheduler could require the predicted runtime for the job with the characterization $\alpha = exec = CPMD, cpus = 12, user = fguim, cwd = /home/fguim/cpmd128$ and the predictor could return $\alpha_{rt} = 120$ minutes.
- Usually, the prediction will not be perfect. In those cases that the prediction is higher than the real runtime, the scheduler can manage the prediction error and the services of the prediction are not required. As was discussed in the introduction, the scheduler will rearrange the schedule of the reservation table based on the new hole that the job has left. However, in cases where the prediction is lower than the real runtime, the scheduler usually requires a new prediction for the job. In these situations the *MissPrediction* event will be triggered to the predictor using the *Deadlinemiss* functionality. It will have to provide a new estimation of job runtime. The predictors will use the components called *Missmanagers* to estimate the new predicted runtime. These components only attempt to provide new estimations for jobs that have been wrongly predicted.
- The scheduler must notify the predictor of all events during the complete job cycle. On job arrival and job start, the predictor will be notified with the time stamp of both events. In the first event type, the predictor may return a job runtime prediction. When a job finishes, the scheduler will notify this event to the predictor using the *JobFinish* functionality, and will provide the real performance variables of the job (such as the real runtime or the real consumed memory).

All the predictors that are available in the simulator have to provide these three functionalities. Similar to the data mining components, we have defined an abstract

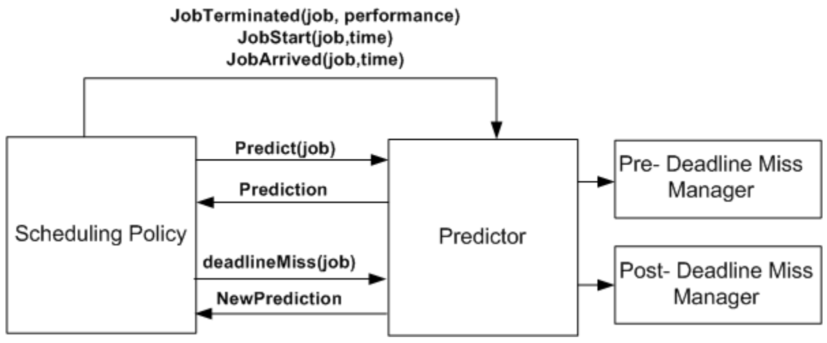


Figure 4.6: General Prediction Internals

interface which all the predictors and missed deadline managers have to provide. Thanks to this model, predictors and *missmanagers* with different properties can be used in the same way in other components of the simulator.

The interface that all the predictors have to implement is presented in the above code. The first three methods deal with three job events : arrival, start and finish (lines 3-5). The *estimateJobPerformance* has to implement the prediction functionality and return a runtime prediction when it is required by the scheduler (or any other component). And finally, the last method that has to be implemented will be called when a missed deadline is triggered by the scheduler.

```
1 Predictor(Log* log);
2 virtual ~Predictor();
3 virtual Prediction* jobArrived(Job* job) = 0;
4 virtual void jobStarted(Job* job, double time ) = 0;
5 virtual Prediction* jobTerminated(Job* job,  double time,
6                                  bool completed) = 0;
7 virtual Prediction* estimateJobPerformance(Job* job) = 0;
8 virtual Prediction* jobDeadlineMissed(Job* job,
9                                  double time) = 0;
```

The following source shows the interfaces that all the deadline managers have to implement. The first four methods allow to get/set information about the characteristics of the deadline manager: if t is a post manager it may be used to generate new predictions when the previous prediction was lower than the user estimate; if it is a pre manager it may be used to generate new predictions when the previous prediction was higher than, or equal to, the user estimate. The last method allows for the gathering of a new runtime prediction for a job that has been wrongly predicted.

```
void setIsPostManager(bool theValue);
bool getisPostManager() const;
void setIsPreManager(bool theValue);
bool getisPreManager() const;
virtual double deadlineMiss(Job* job, double elapsed,
                            double estimate, int miss_count) = 0;
```

### 4.4.2 The available Predictors

The simulator provides a set of predictors that can be used by the simulator schedulers to estimate job runtime (or other performance variables) based on data mining techniques, statistical techniques and regression techniques. Furthermore, they can be used by any other components that require such techniques. The only restriction is that they must follow the interaction described in the previous subsection. The available predictors are:

- The *User runtime estimate* is a simple predictor that returns the runtime estimate that was provided by the user in the job submission. This predictor may seem quite useless, and yet, in some situations (e.g. when there is not enough information to provide a prediction for a given job), its is preferable to use the runtime estimate rather than a random or constant prediction.

- The *Constant predictor* is the simplest predictor that always returns a constant prediction for all jobs. This constant can be specified in the predictor configuration. However, by default the constant is 60 seconds. This predictor was proposed in (92) by Talbi et al. The main goal is to use it when there is no information available to predict job runtime, neither historical information nor a job runtime user estimate.

- The *Historical Classifier* predictor is built on top of discretization techniques and ID3 an C.45 classification trees. This predictor is likely to provide more accurate predictions when the amount of historical information is minimum. It is recommendable to use this simulator together with a simpler predictor (like the constant, or the user runtime estimate). During an initial phase of model warm up, when the historical information is insufficient to provide enough accurate predictions to build up the model, a simpler predictor could be used. Thus, when the historical instance set of information is enough, then this classifier can be used. The techniques designed for this predictor, which were built using the data mining components described in the previous subsection, are described in more detail in Chapter 9.

- The *Regression Model* predictor is based on a linear regression model where job runtime is the predicted variable. Different regression models can be built, depending on the input variable used to compute the regression parameters: the number of processors or the amount of required memory. The predictor also allows for the use of the regression model using simple clustering techniques. The clusters can be built using a user identifier or a group identifier.

- The *Statistical Model* predictor is based on statistical estimators to predict carrying out a prediction. Similar to the previous predictor, a simple clustering technique is applied to all the available instance sets and then a set of statistical estimators are computed to each cluster. Thus, when a prediction is required for a given job, first the cluster that matches the job is found, and then the required statistical estimator is computed in the given cluster and returned as a prediction. As with the *Regression Model*, the clusters are built using a user identifier or a group identifier. In the current version of the simulator, two different of estimators can be queried, the biased estimators mean and standard deviation, and the non-biased estimators median and IQR.

As stated in (92), the most appropriate configuration is to use not only one specific predictor in the system. We agree that a scheduling system which wishes to use job runtime prediction rather than user estimates should use a hybrid combination of different predictors. For example, use a more complex strategy, such as the *Historical Classifier*, when the instance set available is big enough to provide an accurate prediction, and use simpler models, like the *User runtime estimate* predictor when no historical information is available, or use the simplest model when the user runtime estimate is also unavailable.

### 4.4.3 The available Deadline Managers

The deadline managers have the important functionality of correcting a prediction error that a given predictor made in the past when estimating the job runtime of a given job. Based on the experiences outlined in the Tsafrir study, our simulator provides two different kinds of miss managers:

- The *Gradual Deadline Manager*, depending on the amount of missed predictions that the runtime of a the given job has experienced, adds an amount of constant time to the previous predicted runtime. The constants that by default are applied are: 1 min, 5 min, 15 min, 30 min, 1 hour, 2 hours, 10 hours, 20 hours, 50 hours, 100 hours, 500 hours and 1000 hours. Thus, if the job $\alpha$ has been miss-predicted 5 times, on the next missed prediction, the deadline manager would return 2 hours.

- The *Exponential Deadline Manager* adds an exponential amount of runtime based on the previous prediction. The formula applied to compute the new prediction is the following: $p_{\{x,\alpha\}} = p_{\{x-1,\alpha\}}^{\varphi}$, where the factor $\varphi$ has to be provided in the predictor configuration, by default its value is 2.

## 4.5 The Utile Module

This module provides a set of basic functionalities that aim to provide basic services to the other components of the simulator. Basically, the components that are included in this module, do not have enough relevance to be included as an independent module. In this subsection we present all the components that are provided:

- The *Argument Generic Parser* provides functionalities to extract the parameters that are provided to an application or component when it is called. The expected input is a string following the UNIX standard Shell style used to call the binaries and provide the parameters. The outputs for this component are a set of structures containing the parsed arguments.

- The *Configuration File Loader* provides functionalities to validate, parse, and extract information from the simulator XML configuration file. All the functionalities available on the simulator are known by this component. It was built on top of the Xerces Parser Engine (79)(43) and the Xalan XSLT Engine (42) C++ technologies.

- The *CSV Streamers* provides a set of functionalities that allow for the loading, modifying and saving of CSV files. The simulator provides a set of generic interfaces to access the data contained in CSV files and to create new CSV files. It also provides specialized functionalities to create CSV files with specific content. For example, given a set of *Job* and information regarding their characteristics and the performance achieved during their scheduling, the CSV tools can generate a set of CSV files containing all their information.

- The *Job Queues* component provides a set of generic queues that allows for the storing of a set of jobs based on different criteria. The queues are defined as: First-Come-First-Serve order, Estimated Runtime order, Estimated Finish Time order, Real Runtime order, Real Finish Time order and the LXWF. However, following the same generic design principle used in the two previous modules, we defined an abstract interface for the Job Queue. Thus, the researcher is able to implement and include his/her own queues and these can be immediately used by the other components of the system.

- The *Architecture Configuration Loader* component allows for the loading of architecture XML configuration files. As with the previous component, the simulator has a generic interface that defines how the documents have to be defined and accessed, and that provides the more generic functionalities to load some basic elements on the configuration files (i.e.: a node definition or a processors definition). This component includes specific implementations for load configuration files of different typologies of architectures, for example to load an architecture file that defines a Marenostrum type definition.

- The *Predictor Configuration Loader* component allows for the loading of predictor XML configuration files. All the characteristics and properties that are configurable in all the predictors presented previously are extracted using this component.

- The *Statistics Configuration Loader* components allows for the loading of a configuration of statistical components that have to be used during the simulation.

- The *Paraver* component generates Paraver (79) Traces containing the information of how the jobs have been scheduled and allocated in the architecture during the simulation. The Paraver tool will allow to the researcher to visually understand how the scheduling policy has been preceded during the simulation. Thus, using this tool it is easier to detect possible strange behaviors or anomalies in the workflow of the simulations.

- The *Workload Loader* is the component that allows for the loading of job information stored in a workload file. It loads traces that are based on the Feitelson Standard Workload Format (SWF). We created an extension to this format that provides job resource requirement information. Thus, when required, the loader can also load the SWF Extended traces.

## 4.6 The Architecture Module

The Architecture Module is the module of the simulator that contains the definitions of computing architecture models which can be simulated. As with all the components, it contains a set of abstract definitions that can be extended or used in concrete architecture definitions. We have defined the following basic components:

- The *processor* models the computational unit of the system. Using the abstract definition in a model the researcher can specify the processor type (for instance: Power 4 o Power 3), the processor frequency (specified in MHZ) and the power consumption (the amount of megawatts per hour that the processor uses).
- The *node* models a computational node that holds a set of processors. Each of these processors can be homogeneous or the system of node can be composed of a set of different types of processors. The researcher can also specify a set of predefined attributes: the memory bandwidth of the node (in Megabytes/Second), the Ethernet bandwidth of the node (in Megabytes/Second), and the memory of the node (in Megabytes).
- The *blade center* models the computational blade that holds a set of nodes. Each of these nodes can be defined using the previous definitions. Furthermore, the researcher can also specify the memory bandwidth of the node (in Megabytes/Second) and the Ethernet bandwidth (in Megabytes/Second).

Although the core of the module provides these five different abstractions, the researcher can easily extend these definitions or create new architecture definitions. Using these basic components we have provided a set of predefined architectures that the researcher can use in his/her simulation modules. The current version of the simulator provides two different typologies of architectures: a Shared Multi-Processor machine where the computational machine is composed of a set of nodes which share the same memory bandwidth, Ethernet bandwidth and the main memory; and a Marenostrum based architecture. The MareNostrum is a supercomputer based on PowerPC processors, the architecture Blade Center, a Linux system and a Myrinet interconnection. Figure 6.1 provides a description of a possible instantiation of the concrete Marenostrum Architecture model. All the research studies in this Thesis have used this model.

As will be discussed later, the simulation engine and the reservation table may use these models to carry out the scheduling and the modelization of how the jobs are allocated in the simulated architecture.

## 4.7 The Simulation Module

The simulation module is the main module of the simulator. It contains the models that are used in the simulations and the two components that are responsible for coordinating

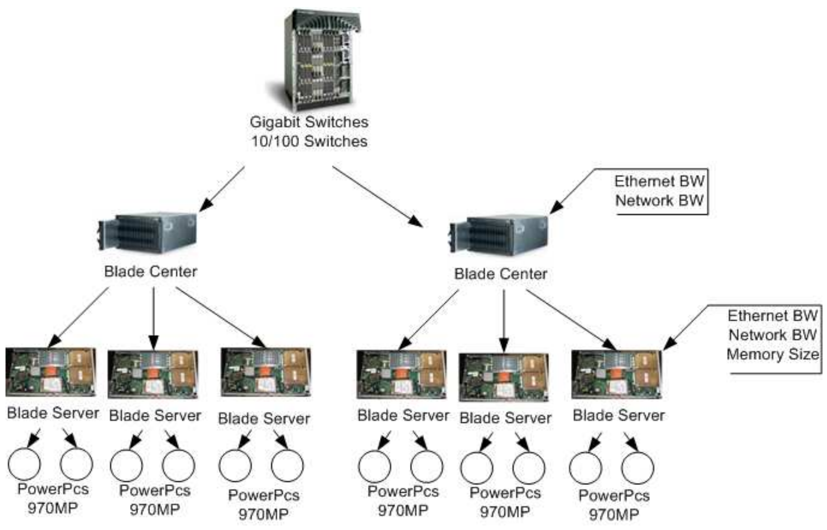


Figure 4.7: Example of Marenostrum Architecture

and carrying out the main actions during the simulation. In this section we provide descriptions of the internal structures of these components. However, we do not provide detailed characterizations of the different models that are included. More information can be found in other sections of the document.

Figure 4.8 shows the minimum components that are instantiated in a given simulation inside the *Simulation Module*. The element that orchestrates all simulation actions is the *Simulator Engine*. It is responsible for loading the configuration fields which are the input of a simulation and instantiate all the different components that are required to carry out the simulation. Furthermore, it controls all the simulation workflow including job event scheduling, information collection regarding the system status and the final processing of all the simulation information necessary to generate the final reports. The other basic elements that are always present in the module are the scheduling policy, the resource selection policy and the reservation table. The Scheduling Policy is the component that models the scheduler that would be installed in a given host of a given center. It receives the jobs submissions from the *Simulator Engine* as they would be done by systems users, and schedules the jobs according to the job scheduling policy that it models (i.e: FCFS or Backfilling). When the Scheduling Policy decides that a given job has to be allocated to the simulated host, it contacts the *Local Resource Manager* component. This component models the typically Local Resource Manager that is installed in the host (i.e: LoadLeveler or SLURM) and decides, based on the

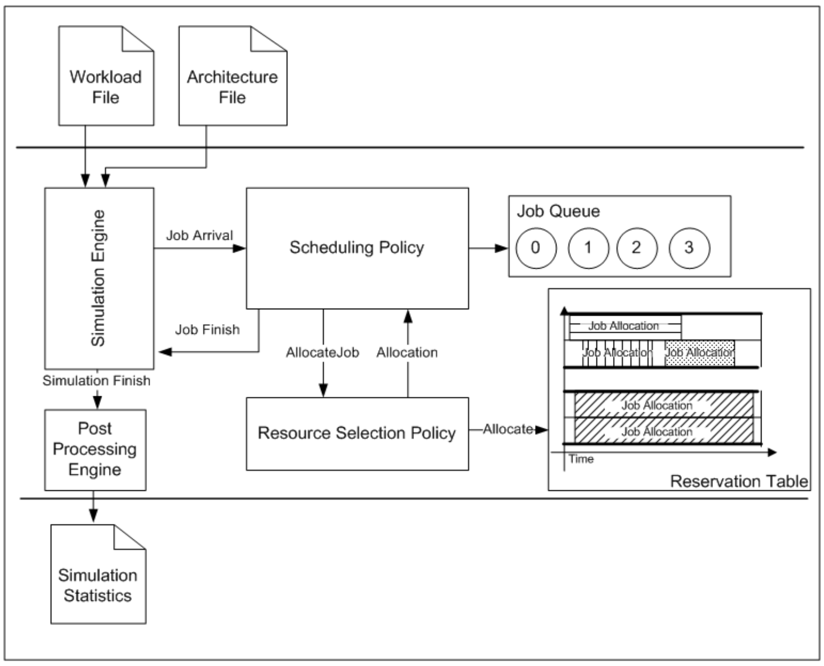


Figure 4.8: Basic Simulation Environment

selection policies that it models (i.e: first fit), which processors the jobs have to be allocated to. The *Resource Selection* thanks to the *Reservation Table* knows the status of the processors, how they are being used and, depending on the simulations settings, the possible outcome of the use of the processors in the future. The *Reservation Table* is one of the most important elements of the simulator. It models how the jobs are allocated to the processors and how the resources are used during the simulation. To the best of our knowledge the Alvio simulator is the first workload simulator that can model how resources are used by all jobs at every moment of a simulation.

A more complex scenario is presented in figure 4.10. This scenario presents a modelization of a meta-scheduler scenario. The figure presents the different components that are instantiated inside the simulator in this kind of scenario. Firstly the meta-scheduler entity is created. In this case the global meta-schedulers will be a set of ISIS-Dispatcher components. Note that the grid-backfilling policy could also be instantiated. Secondly, the prediction system with the data mining module and the pre/post managers are created. They will provide the scheduling entities with functionalities to estimate job runtime variables. The different centers of the local-scenarios of the simulation model

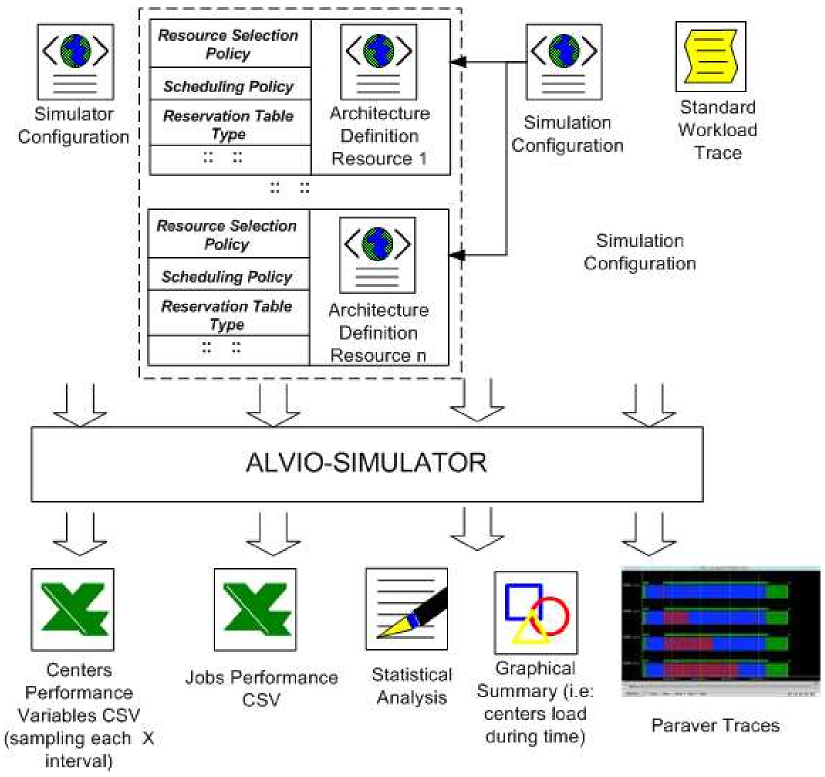


Figure 4.9: Complex Simulation Environment Simulation Inputs/Outputs

are also instantiated. For each of the center components a local scheduling policy (such as the First Come First Serve or the EASY-Backfilling), a resource selection policy (such as the First-Fit or the Find-LessConsume) and the reservation table are created. Finally, all the components that process the simulation statistics regarding the performance of the system are instantiated. These components include: the post process engine that will collect the final information from the Job and Policy entities data collectors, the CSV Collectors, that will dump all the information from the data collectors to the CSV files specified in the configuration files, and the R module, that will process the different CSV files generating the analysis that has been specified by the researcher. Figure 4.9 shows the different input configuration files that have to be provided to the simulator in order to carry out a simulation of the scenario described.

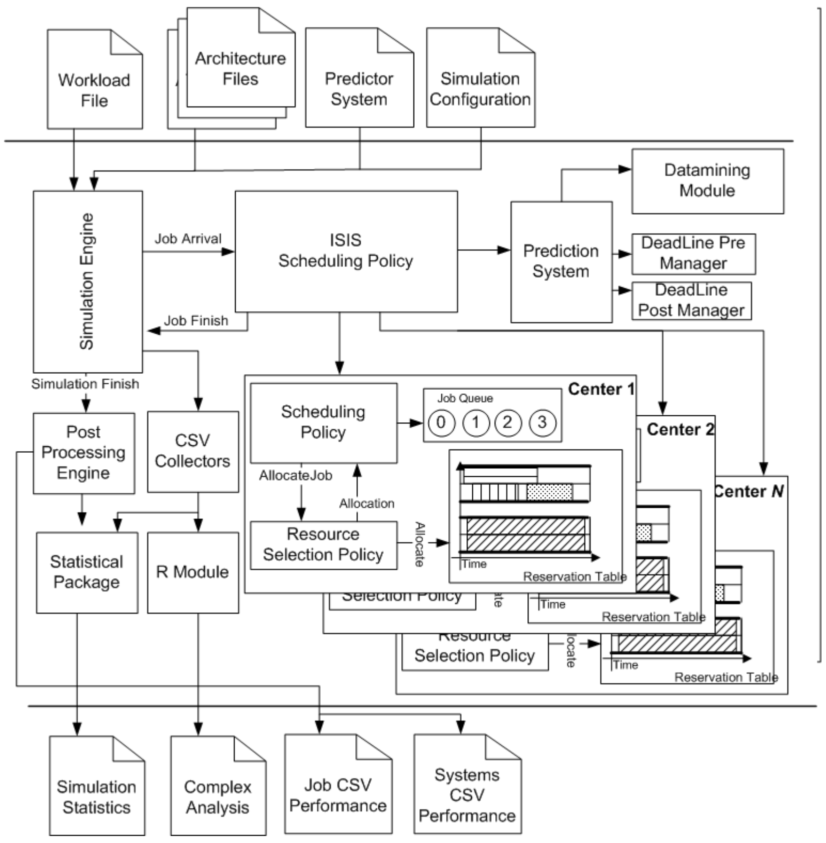


Figure 4.10: Complex Simulation Environment

### 4.7.1 The Simulator Engine

As we have already explained, the *Simulator Engine* is the component of the simulator that is responsible for instantiating and coordinating all the other components of the simulator. Furthermore, it is the responsible for starting, controlling and finishing all the steps in the simulation. The steps that this Engine follows during the simulator life are:

#### 4.7.1.1 Pre Simulation Steps

After starting the simulation the Engine has to set up all the architectural components that are involved in the simulation. The steps that it follows are:

1. Using the *Simulation XML Configuration Loader* it loads the configuration of the

simulation. (The configuration file is provided as an input file).

2. In cases where the Scheduling Policy to simulate is a non distributed architecture:
   (a) It extracts the architecture file that has to be simulated, creates the *Architecture XML Configuration Loader* and asks the loader to instantiate the architecture model to be simulated.
   (b) It creates an instance of the scheduling policy required in the simulation. For example, a First-Come-First-Serve Scheduling Policy. Each Scheduling instance will create the *Reservation Table* it requires for the simulation (depending on the policy the reservation table models used may differ). If the prediction service is required to be used in the configuration:
      i. It extracts the predictor configuration file, creates the required *Prediction Module* required, and provides the reference of the prediction service to the Scheduling Policy. The *Prediction Module* may instantiate other components, for instance the Data mining module or the Statistical module, depending on the prediction service that has to be instantiated.
3. In cases where the Scheduling Policy to simulate is a distributed architecture, composed of several centers with several Architectures, Scheduling Policies, and Resource Selection Policies:
   (a) It creates an instance of the distributed scheduling policy required in the simulation.
   (b) It creates all the centers with its scheduling policies, resources selection policies and architecture model in the same way as was described for the non-distributed architecture. However, depending on the policy (i.e: the Grid-Backfilling Policy) for each center only the resource selection policy and the architecture module will be instantiated.
4. Using the appropriate *Workload Loader*, it create the whole punch of jobs that are included in the trace file.
5. It creates the *Statistics Configuration Loader* and loads the statistics configuration file specified in the simulation. In this configuration file, the researcher must specify which statistics have to be computed for the job performance variables and which for the policy (or policies in the case of a distributed architecture). Once the statistical configuration information is loaded it will create the *Statistical Module* and the required data collectors.

6. In specified cases in the simulation configuration, it will create the appropriate *CSV Streamers*. Usually, it creates one for the job performance information and one for the policy/ies performance information.

7. In specified cases a *Paraver* component will be created. During the simulation the information that the component requires will be provided.

8. Finally, it creates all the simulation events necessary to start the simulation:

   (a) First the event arrivals for all the jobs of those included in the workload will be created.

   (b) Second, the first collect statistic event will be created. In the event queue there is only one event of this type and it is triggered each $\phi$ simulation clocks. When this event is processed, the status of all the systems of the architecture (i.e: number of queued jobs, number of processors used etc.) are collected and stored in the appropriate CSV Streamer.

#### 4.7.1.2 Simulation Steps

The simulator iterates over an infinite loop while there are simulation events in the simulator event queue. There five different events that the engine has to deal with:

- The *EVENT_TERMINATION* or the *EVENT_ABNORMAL_TERMINATION* are triggered when a job is finished. When this event is dealt with two major actions are performed: first the prediction service is informed of its termination and the performance information is provided (in cases where a prediction service is used); and second, the appropriate data collector is updated with the job performance information.

- The *EVENT_START* is triggered when a job starts to run in the computational resource. In this case the prediction service is informed about this event(in cases where a prediction service is used). A *EVENT_TERMINATION* event will be created for this job.

- The *EVENT_ARRIVAL* is triggered when a job is submitted in the system. The job submission is forwarded to the scheduling system in the same way as a system user would do. When the scheduling policy decides when the job will start the *EVENT_START* for the job will be created. It depends on the scheduling policy that is being simulated.

- The *EVENT_COLLECT_STATISTICS*, as has been mentioned before, is triggered each $\phi$ simulation clocks. When this event is processed the performance variable of the center/centers of the simulation environment is gathered using the appropriate data collector. A new *EVENT_COLLECT_STATISTICS* is generated for the next $\phi$ simulation units.

#### 4.7.1.3 Post Simulation Steps

When all the events of the simulation have been processed, the simulation is considered to be finished. Thus, the only step that remains is to create a simulation configuration file which contains the following output information:

- In cases where the *Paraver* component was created, the paraver trace files will be dumped according to the simulation.
- It provides the CSV Streamers with the reference to the appropriate data collectors and requires them to generate the appropriate CSV files.
- It extracts all the *R Analyzers* that are specified in the configuration file, instantiates them and requires them to carry out the analysis in the CSV files that have been dumped in the previous step.
- It extracts the data collectors to the *Statistical Module* and requires it to carry out all the statistical analyses on the collected data. Basically, it applies all the estimators that are specified in the statistics configuration file to all the variables that are specified in the same file. Once the estimators are computed their results are dumped in the output files.

### 4.7.2 The Scheduling Policy

This component models the typical Job Scheduling Policy of high performance computing centers. When simulating a non-distributed architecture, the simulator only has one instance of this component. In a distributed system many instantiations may be included in the same simulation environment. Obviously, the simulator provides different types of scheduling instances. As with the *Data mining* and the *Prediction Module*, we have defined a common abstract interface which all scheduling policies have to provide. Thus, the *Simulation Engine* is not aware of what type of policy is being simulated, since they all implement the same interface. The methods that a policy has to implement are shown in the following source. Note that the API defined here are partially based on the interface proposed by Tsafrir in the simulator developed in (96):

- The *jobArrival* method is called when a job is submitted to the system. Depending on the policy that it models, the scheduler may decide to start the job or to add it to the wait queue.
- The *jobAdd2waitq* method is usually called by the same *Scheduling Policy* when the method *jobArrival* is executed.
- The *jobWaitq2RT* method is called by the simulation engine when a *EVENT_START* event is raised. The scheduler will start the job in the given processor allocation that was decided by the resource selection policy when the job was chosen to start by the same scheduler.
- The *jobRemoveFromRT* method is called by the *Simulation Engine* when an event of *EVENT_TERMINATION* or the *EVENT_ABNORMAL_TERMINATION* is raised.
- The *jobChoose* method, usually called by the same scheduling policy, given the job wait queue and the status of the *Reservation Table* decides which job has to start.
- The methods *getJobsIntheWQ, getBackfilledJobs, getLeftWork and getJobsIntheRQ* are used by the *Policy data collectors* to retrieve information concerning the status of the system at any given point of time. Usually, these methods are required by the collector each time the *EVENT_COLLECT_STATISTICS* event is processed.
- Finally, the method *EstimatePerformanceResponse* is responsible for providing estimations of a certain metric type given the characteristics of the provided job. For example, it could be required to estimate the wait time that a given job would experiment if it were submitted at a given point of time. Obviously, the *Scheduling Policy* is not required to provide all the metric estimations. This method was designed for use in distributed architectures where there are scheduling entities running on the top of the architecture. The *Schedulers* that schedule the jobs on the top of these systems are available to decide where to submit the job, based on the estimations that the local policies provide with this method. More detailed information about these models can be found in the section describing the ISIS Scheduling Policy.

```
SchedulingPolicy();
virtual ~SchedulingPolicy();
virtual void jobAdd2waitq(Job* job);
virtual void jobWaitq2RT(Job* job) = 0;
virtual void jobRemoveFromRT(Job* job)= 0;
virtual Job* jobChoose() =  0;
virtual void jobStart(Job* job) = 0;
virtual double getJobsIntheWQ() = 0;
virtual double getBackfilledJobs();
virtual double getLeftWork() = 0;
virtual double getJobsIntheRQ() = 0;
virtual Metric* EstimatePerformanceResponse
                        (metric_t MetricTpye,Job* job) = 0;
```

The simulator provides implementations of eight different *Scheduling Policies* assigned to four different groups. The first group contains the backfilling based scheduling policies, the second group is the typically First-Come-First-Serve scheduling policy, the third group contains the centralized distributed scheduling architectures, and finally, the last group are made up of the distributed non centralized scheduling architectures. The policies that are available (shown in figure 4.11) are:

- The backfilling policy has been implemented. The traditional backfilling policy that is usually simulated in experiments has one reservation, and the wait queue and the backfilling queue are ordered in the first come first served order. However, the simulator can use all the different backfilling variants. Thus, the researcher is able to specify how many reservations have to be used, from $[1..n)$, the order in which the jobs have to be ordered in the wait queue (this order determines in which order the jobs are assigned to the free reservations) and in which order the jobs are backfilled. For more information about the different backfilling variants, see the introduction section. Two variants implemented in this policy are implemented in the simulator:
  - The first variant allows for the use of a predictor service in the simulation. Thus, the scheduler schedules the jobs using any of the previous backfilling variants but uses the runtime predicted provided by the predictor rather than the user runtime estimates provided in the workload. For more information see the ”‘Prediction Models”’ chapter.
  - The second variant allows a simulation of the backfilling and variants, but models how the resources are used during the simulation. In this variant, the backfilling simulations use the *Reservation Table* that models resource usage and job requirements. Thus, the jobs may have runtime penalties depending

on where and when they are allocated, and the status of the system in their allocation slots. For more information about this policy see the section ”‘Modelling the Resource Usage with the Alvio Simulator”’.

- The RUA-Backfilling policy implements a backfilling based scheduling policy that considers resource sharing when carrying out the job allocation and backfilling process. For more information see Chapter 7.
- The first-come-first-serve policy provides a model for the simplest scheduling policy where jobs are started to run in the order in which they arrive. In some situations, the researcher may decide to use this policy to compare situations with close to optimum situation with the worst performance situation.
- The ISIS-Dispatcher allows the simulation of a non-centralized scheduling policy for distributed architectures where the jobs are scheduled by independent scheduling entities. More information about this policy can be found in the section ”‘Self Scheduling Policies for HPC Infrastructures”’. The simulator allows the use of a variant of this policy, where the job runtime of the job in a given center can be estimated using a predictor service.
- The Grid-Backfilling scheduling policy allows the simulation of a centralized scheduling policy for distributed architectures where the jobs are scheduled by centralized schedulers using a global backfilling algorithm. As with the previous policy, it can use a prediction service to estimate the job runtime in a given computational resource. More information about this policy can be found in (59).

### 4.7.3 The Resource Selection Policy

Given a job with a set of computational requirements, the Resource Selection Policy (RSP) decides which processors it has to be allocated to. As with the Scheduling Policies, the simulator provides different RSPs that can be used by the researcher in combination with different job scheduling policies. Since all the selection policies are implemented following the same interfaces, they can be used regardless of any source code modification. The following source shows the interface that a given resource selection policy has to implement:

```
AnalogicalJobAllocation* findAllocation(Job* job,
                         double starttime,
                         double runtime) = 0;
```

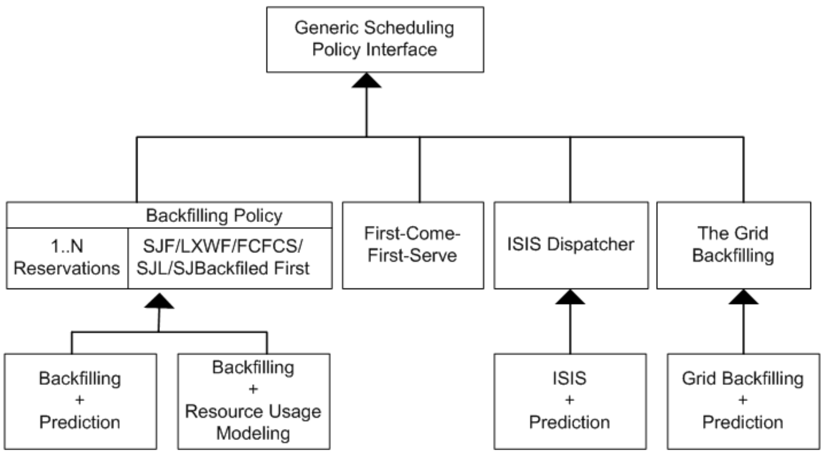

Generic Scheduling Policy Interface
Backfilling Policy
1..N Reservations
SJF/LXWF/FCFCS/ SJL/SJBackfiled First
First-Come-First-Serve
ISIS Dispatcher
The Grid Backfilling
Backfilling + Prediction
Backfilling + Resource Usage Modeling
ISIS + Prediction
Grid Backfilling + Prediction

Figure 4.11: Scheduling Policies

```
bool allocateJob(Job* job,
                 JobAllocation* genericallocation) = 0;
bool deAllocateJob(Job* job) = 0;
Metric* getRTablePerformanceMetric(metric_t mType) = 0;
```

Given a job characterization, including its statistical requirements, such as the number of requested processors, and its resources requirements, including the required memory bandwidth, the first method is designed to find an allocation where the job can run. An allocation is composed of a start time, a finish time, and a set of processors. Clearly, how the allocation is chosen totally depends on the resource selection policy that this RSP component implements. Moreover, the returned allocation start time may be a future point of time. For example, this could occur when the requirements for the application (i.e: the number of required processors) cannot be satisfied until this future point of time. The *findAllocation* method is usually required from the Scheduling Policy when it is scheduling a job that has to be allocated, or that has to check when it should start. The second method is required when the Scheduling Policy has decided that a given job has to start at a given allocation. Thus, the RSP will allocate the job processes to the processors specified in the allocation. The third method does the contrary: once the job is finished or has to be killed by the scheduler, the Scheduling Policy requires the RSP to deallocate the job from the processors where it is running. The last method returns information about the status of the reservation table that is used by the resource selection policy.

The simulator provides seven different resource selection policies:

- The *first fit* policy is the simplest selection policy. It finds the first $1..N$ processors that allow the earliest start to the processes of a given job. Note that the processors will probably not be consecutive. Since the policy does not have to consider any selection restriction, it will always return the earliest possible allocation.
- The *first continuous fit* policy will return the first $1..N$ consecutive processors that allow the earliest start to the processes of a given job. Note that there is probably a first fit processor selection that would start earlier than that obtained with this selection policy.
- The *first fit or first continuous fit* policy will return a first continuous fit based selection. However, if a first continuous fit based selection returns an earlier allocation, the latter will be returned.
- The *find less consume* policy will return the first $1..N$ processors that satisfy the condition that, given the provided resource requirements of the job and the current status of the system, the job experiences the lowest penalized runtime. For more information see Chapter 7.
- The *find less consume threshold* policy will return the first $1..N$ processors that satisfy the condition that, given the provided resource requirements of the job and the current status of the system, the job experiences a lower penalized runtime than a provided threshold. Note that in this situation the returned policy is probably not the one with the lowest penalized runtime. For more information see Chapter 7.
- The *Equi-distribute consume* policy will return the first $1..N$ processors distributing the processes among the different nodes of the host. The main goal of this policy is to avoid evaluating the different possible allocations based on the resource requirements of the job. In this situation the selection is made by a static selection.

### 4.7.4 The Reservation Table

This component is one of the most important components of the simulator and it is used by the resource selection policies to allocate jobs on the architecture and to evaluate how possible allocations behave. The reservation table models how the processors and resources are used in the simulated architecture. As is shown in figure 6.2, the reservation table contains the mapping of the different jobs that are currently running in the architecture (*Job 1*,*Job 2* and *Job 4*) and the future allocations for the jobs that

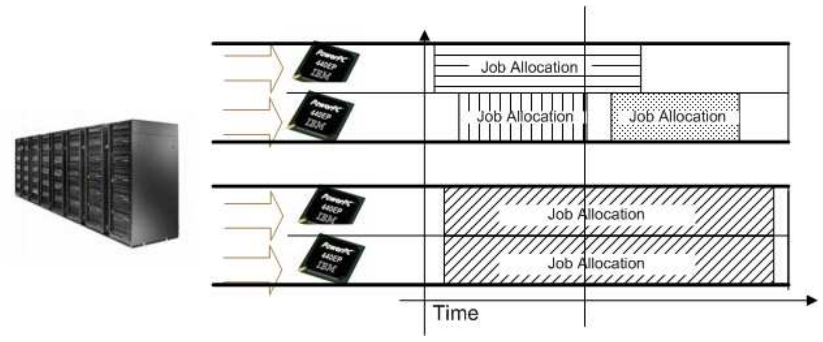


Figure 4.12: The reservation table

are scheduled (*Job 4*). This last type of allocations depends on the scheduling policy and resource selection policy that is simulated, not all of them use reservations for the processors.

As with the previous components, we have defined a set of abstract methods that a reservation table must provide. Thus, the resource selection policies can use different kinds of reservation tables with different characteristics using the same interface. The source code presented below shows the interface that all the reservation tables must provide. They provide four different functionalities:

- Methods to allocate and deallocate jobs to different processors in the current simulation time. The resource selection policies can start jobs *now* in the specified processors and can free resources for the jobs that are finished. (lines 5-8)
- Methods to extend or reduce the amount of time for jobs that are currently running. (lines 9-10)
- Basic methods to determine intervals of time where different processors are free. These methods will be used by the resource selection policies to decide where to allocate different jobs. New definitions of reservation tables may provide other methods to decide on these intervals based on specific criteria. For example, returning the processors for those nodes whose resources are less loaded.

```
1 Reservation Table();
2 ReservationTable(ArchitectureConfiguration* theValue,
3                  Log* log,
4                  double globaltime);
```

```
virtual ~ReservationTable();
void setArchitecture(ArchitectureConfiguration* theValue);
ArchitectureConfiguration* getarchitecture() const;
virtual bool allocateJob(Job* job,
                        JobAllocation* allocation) = 0;
virtual bool deAllocateJob(Job* job) = 0;
virtual bool killJob(Job* job) = 0;
virtual bool extendRuntime(Job* job,double length);
virtual bool reduceRuntime(Job* job,double length);
BucketSetOrdered findFirstBucketCpus(double time,
                                     double length);
```

The simulator provides three different types of reservation tables:

- The *Analogical Reservation Table* implements the reservation introduced in figure 6.2. It has the running jobs allocated to the different processors during the time. One allocation is composed of a set of buckets that indicate that a given job $\alpha$ is using the processors $\beta$ from $T_{startTime}$ until $T_{endTime}$.
- The *Virtual Reservation Table* extends the model used in the Analogical Reservation Table and considers how the resources of the computational resources are being used (see figure 4.13). The formalization of this reservation table is introduced in the section ”‘Modeling the Resource Usage with the Alvio Simulator”’.
- The *Grid Reservation Table* following the Analogical Reservation Table model implements a model that manages all the processors available in a distributed architecture. A more detailed description of this model is presented in (59).

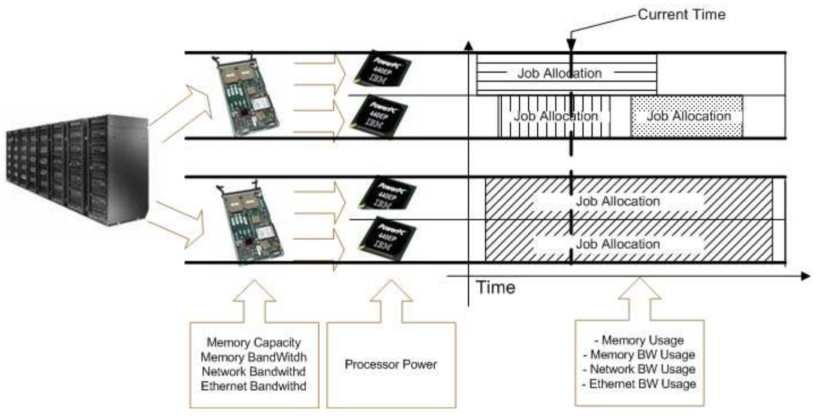


Figure 4.13: The reservation table with resource usage

Chapter 5

# Prediction *f* based Models for Evaluating Backfilling Scheduling Policies

In this chapter we describe and evaluate a set of f-model based prediction models which characterize the behavior of prediction techniques used in HPC centers. The models have been designed to evaluate scheduling policies that use predictions rather than user estimates. Furthermore, we provide an evaluation impact of these models to a set of widely used backfilling variants.

## 5.1 Introduction

Backfilling based scheduling policies have achieved the highest performance results in parallel HPC architectures. However, the major inconvenience of their base algorithm is that scheduling is based on the job runtime estimation that users provide. Thus, the runtime of the scheduled applications has to be known at submission time.

Several studies have investigated the impact of user estimation accuracy on the performance of these scheduling policies. The commonly used f-model has been used to evaluate how the accuracy of user estimations affect the performance of backfilling and variants. As a result of such analysis, researchers have proposed new backfilling variant scheduling policies, as S-H Chiang did with LXWF-Backfilling or Tsafrir did with Shortest Job Backfilled First (SJBF).

What has become increasingly relevant in the last few years is the use of prediction techniques in scheduling policies rather than user estimates. In emerging scheduling architectures, such as Grids and very heterogeneous computational resources, these techniques are crucially relevant since, in most cases, users will not have enough information or enough skills to specify how long their jobs will run. Some researchers have started to face this complex problem. Tsafrir et al. have provided the community with two papers that study the impact of the use of four different prediction techniques rather than user estimates in backfilling policies (98). A similar study is presented by Talbi et al. in (92). They evaluate the impact of the use of three different predictors for the backfilling policies that are designed to work with different amounts of information at the prediction time. All these papers have analyzed the impact of specific predictors with specific properties on these backfilling policies.

The traditional f-model cannot be used in the evaluation of the performance of scheduling policies which use prediction techniques rather than user estimates. As will be discussed later, this model does not properly model some characteristics inherent in the prediction techniques which can have important effects on the behavior and performance of the studied scheduling systems. This is mainly due to the essence of the predictor properties, which are clearly different to those that users usually show in their runtime estimations. For example, while users usually tend to overestimate their jobs by a factor of 2, or even 5, to avoid their jobs being killed, a predictor can underestimate the job runtime by a factor of 100. Obviously this is because real users have some clues concerning how they applications will behave, while the predictor does not. In this paper we define a set of models which have several properties inherent in the predictors' estimations that can have an impact on the performance of the system. This characterization is based on our research background in prediction and scheduling techniques (38)(54)(58)(71)(48)(59).

We have analyzed the impact of these models on different sets of workloads available in (28). In the experiments we have seen that adding or subtracting quantitative errors to all the runtime jobs predictions of the workload have no relevant impact on the performance of the system. Furthermore, adding or subtracting quantitative errors to a determined subset of jobs does not have any relevant impact on the performance of the system. The only subset of jobs that has shown a real impact on performance is short jobs. We have also demonstrated how qualitative errors on runtime predictions can result in a dramatic drop in performance. Thus, predicting that a job will be short while it is really large, or vice versa, can substantially increase performance variables such as the average bounded slowdown.

The rest of the chapter is organized as follows: sections 5.2 and 5.3 we present the related work and our main contributions; section 5.5 presents the prediction models that are described; sections 5.6 and 5.7 the experiments and the evaluation of the models are presented; and finally, in section 5.8 we present our conclusions.

## 5.2 Related Work

Researchers have modeled logs collected from large scale parallel systems in production. They have provided knowledge for the evaluation of the different systems behavior. Such studies have been fundamental since they have allowed to understand how the HPC centers users behave and how the resource of such centers are being used. Thereby, with the characterization about the jobs that users submit to the center, the researchers have been able to understand the performance of the different scheduling policies that they designed.

Feitelson has presented several works concerning this topic. Among others, he has published about logs analysis for specific centers (32)(67), general job and workload modelitzation (33)(30)(35)(31), and together with Tsafrir, about detecting workload anomalies and flurries for filter them from the workloads (100)(99). Calzarossa has also contributed with several workload modelitzation surveys (13)(14)(15).

In the context of workload modeling, there is a relevant topic that has also been deeply studied in these previous decades, and that has provided new inputs for the evaluation of scheduling policies: the generation of synthetic workloads. These works have allowed to generate large synthetic workloads with specific properties for evaluate new scheduling strategies. Generally, the models were constructed by collecting real work logs form different HPC centers, removing those jobs that are likely to be flurries, analyzing their properties and characterizing its properties.

Several works have provided models that have been widely used in the performance evaluation research. For instance, a characterization of moldable jobs was proposed by Cirne at al. in (18)(19), or by Sevci in (87) taking into account aspects like imbalance and parallel overheads. Furthermore, Downey also provided a characterization of this type of jobs in (22) modeling their parallelism, the runtime, the speedup and the arrivals. Concerning rigid jobs, Feitelson characterized a model based on observations from 6 logs in (35). The model includes the distribution of job sizes in terms of number of processors, the correlation of runtime with parallelism, and repeated runs of the same job. Also, Lublin proposed a very detailed model in (74) for the same jobs. His model include their arrival pattern and runtime patterns.

In the area of workload modeling, there is a recent work, done by Tsafrir et al. that has been used for evaluating the impact of the user estimates in the performance of the backfilling scheduling policies. Rather than modeling the job profile, their work presents a model characterizing the user runtime estimates for the jobs that are submitting to the centers. This model has a special relevance for the research of these policies since the runtime estimate accuracy has an important effects of how the scheduling is carried out.

The other outstanding topic in the area of HPC infrastructures has been the evaluation and design of scheduling policies. The gang scheduling (30) and the backfilling policies have been the main goal of study these last years (88)(96)(17). In (88) Skovira et. al presented the first paper about the EASY algorithm and its performance in the LoadLeveler system. General descriptions about the most used backfilling variants and parallel scheduling policies can be found in the report that Dror. G. Feitelson et al. provide in (36). Moreover, deeper description of the conservative backfilling algorithm can be found in (91), where the authors present it characterization and how the priorities that can be used when choosing the appropriate job to be scheduled.

Tsafrir et al. have presented a work analyzing the impact of the usage of specific prediction techniques rather user estimates in the backfilling policies (98). In this work they proposed a set of interfaces that the backfilling policies should incorporate to support the usage of prediction techniques rather than user estimates. The main objective of such interfaces is to deal with the deadline misses that are triggered once the job runtime exceeds the runtime prediction that was provided by the prediction system. In the traditional systems the jobs would be killed. However, in this extensions the predicted runtime is extended and the scheduling of the queued jobs is modified. Similar work is the once presented by Talbi et al. in (92) where the impact of the usage of three different predictors to the backfilling policies with the Tsafrir proposed extension is studied (choosing the appropriate once is done depending on the amount of information available as an input for the prediction).

## 5.3 Our Contribution

In this chapter we present a detailed study of the impact of prediction errors on scheduling policies. Clearly, our work has focused on several issues that have not been modeled in previous studies. The traditional user runtime *f-model* used in backfilling scheduling policy evaluation is defined as follows: $est_\alpha \in U\left[r, r*K\right]$. Where the $est_\alpha$ is the runtime estimate for the job $\alpha$ provided by the user, and follows a normal distribution starting at the real runtime $r$ of the job, and finishing at $r*K$ [1]. However, as we have already indicated, it does not consider some properties of predictions. In this study we characterize a set of prediction models which extend the *f-model*:

- We have used models where the prediction of runtime can be lower than the real

[1] $K$ is a constant value higher than 1.

runtime. The estimation f-model does not take into account the fact that jobs can be underestimated since jobs would be killed by the schedulers.

- The other studies used simple error models, since in the experiments the same estimation accuracy model is applied to all jobs. We have evaluated different profiles of errors. By profile, we mean applying errors to jobs with a certain characteristic, for instance: those jobs that use executables which are most frequently submitted to the center, or those jobs that, with respect to the workload used in the simulation, require a high number of processors, etc. Here, we have studied the potential use of specific predictors specialized in certain job typologies.

- We have evaluated the impact of changing the nature of the job on the prediction using qualitative errors. For example, a qualitative error is to predict that the runtime of the job is short while, with respect to the simulated workload, the job was, in fact, large. In this scenario, we also have analyzed the potential use of categorical predictors in the context of backfilling policies.

- As proposed in (98), we have extended backfilling interfaces to treat the missed deadline event for a job. This is triggered when a job has used up all the runtime that was predicted for it when it was submitted. In these situations, in contrast to the approaches that are taken in the traditional studies, the job cannot be killed since the error has not been generated by the user. In our work what the scheduler does is to extend the prediction and the job allocation by an amount of time.

## 5.4 Simulation Framework

All the experiments presented in this chapter were conducted using a C++ event-driven simulator that was implemented and used by Tsafrir et al. in the paper (98). This is a modular simulator that allows us to simulate the EASY-Backfilling and SJBF policies. They are implemented by extending the traditional EASY algorithm to the use of runtime prediction rather than user estimates. It also allows adding predictors modules that are used by the scheduling policies to schedule the job according to its predicted runtime. For the work presented in this paper we have extended the simulator implementing the other backfilling variants and implementing the *fake predictor*. It predicts job runtime by carrying out numerical transformations on the real runtime (presented in section 5.5). We called it **‘fake’** since its predictions are computed by adding or subtracting a value to the real runtime of the job taken from the workload. When we carried out this analysis, the Alvio simulator was not still ready. However its current architecture provides the required functionalities to evaluate the models presented in this chapter.

## 5.5 The Prediction Models

The prediction models that have been characterized are divided into two main groups: quantitative prediction errors and qualitative prediction errors.

### 5.5.1 Prediction Generation Using Quantitative Errors

Quantitative errors are generated by adding or subtracting a percentage to the real runtime of a job and returning it as a prediction. As has already been claimed above, this model is substantially different form the f-model, since the interval for the predictions is $[max(1, r - K * r), r + K * r]$, while the f-model is $[r, K * r]$. We have also evaluated the effect of adding this kind of errors to different sets of jobs:

1. **High number of processors**: Jobs that use more processors than the *upper bound* of the *number of processors* variable [2](see table 5.2).
2. **Low number of processors**: Jobs that use less processors than the *lower bound* of the *number of processors* variable (see table 5.2).
3. **Large jobs**: Jobs whose runtime is greater than the *upper bound* of the *run-time* variable (see table 5.1).
4. **Short jobs**: Jobs whose runtime is less than the *lower bound* of the *run-time* variable (see table 5.1).
5. **High area**: Jobs whose runtime multiplied by the number of processors that it uses is greater than the *upper bound* of the *Area* variable (see table 5.3).
6. **Low area**: Jobs whose runtime multiplied by the number of processors that it uses is lower than the *lower bound* of the *Area* variable (see table 5.3).
7. **More executed applications**: Jobs whose submitted application is one of the most executed in the whole workload. We have defined these applications as the applications that are executed more than 50 times. Each workload contains around 150 of these applications.

[2]The jobs are categorized based on a set of upper and lower bounds that define which jobs matches the descriptions. The computation of these bounds is introduced later and depends on the workload used in the evaluation.

8. **Low processors and low runtime**: Jobs whose runtime is less than the *lower bound* of the *run-time* variable and whose number of used processors is less than the *lower bound* of the *number of processors* variable.
9. **Not small jobs**: Jobs whose runtime is more than the *lower bound* of the *run-time* variable or whose number of used processors is more than the *lower bound* of the *number of processors* variable.
10. **Pure Equal**: All the jobs are predicted using the error and standard deviation that is evaluated in the experiment.
11. **Equal**: A fixed percentage of jobs are perfectly predicted. They are chosen using a random uniform variable. This means that there is no fixed criteria for selecting this subset of jobs. The rest of the jobs are predicted using a provided error and stdev.

In the equal category approximately 50% of the jobs are predicted with accuracy. In the first approach we applied the error to 100% of jobs (pure equal category), however later on we found that this approach was quite pessimistic. We decided to apply the error to 50% percent of the jobs (equal category). Our approach was based on two different considerations. The first one is presented in (17), where it is concluded that when 60% of the jobs provided approximately accurate requested runtimes, the scheduling policy performance was improved. The second consideration was that when we used the other categories, approximately 30-55% of the jobs where predicted with accuracy.

### Upper and Lower Bounds

The application type definitions are based on lower and upper bounds that are computed for each workload and variable (runtime, number of processors and area). Therefore, the application types may differ in different centres, for example, the lower runtime bound for the LANL workload is 224.27 secs. while in the CTC is 1152.2 secs. Initially our definition of these boundaries was based on the statistical properties of the workload variables. For each of the workload variables we computed its percentiles and defined the lower bound as the 20th percentile of the variable and the upper bound as the 80th percentile. However, the experiments that used these bounds did not provide interesting results. This was because the category definitions were based on statistical properties rather than on a logical criterion. For example, using the first criterion the definition of small jobs for the CTC center were all the jobs with runtime less than 10 minutes, but jobs with runtime less than 1 hour were still small compared to the rest of the workload.

To solve this problem we took a similar approach to that taken by Sui-Chiang et al. in (17). We defined the boundaries following a reasonable criterion rather than a statistical one, using as a basis for our decisions the percentiles for each variable for each workload. The upper bound of the variable *var* is defined as the $UpperBound_{var} = 95_{th}Percentile/2$ and the lower bound as $LowerBound_{var} = UpperBound_{var} * 0,05$, where *var* is the workload variable and can be: runtime, number of processors and area.

**The prediction:**

The inputs for the experiments when evaluating the impact of quantitative errors on the scheduling are:

1. The **policy** and **workload trace** used in the simulation (EASY-backfilling, SJBF or LXWF-Backfilling).

2. The **error** and **standard deviation** used in the prediction generation. The deviation is used so as not to apply exactly the same error in all the predictions. Thus, we can also evaluate the effect of having high dispersion in the prediction errors.

3. The **type of jobs** that are predicted with high accuracy (introduced in the first part of the section). When the predictor is asked for a prediction, if the job whose prediction is computed matches this type, a perfect prediction will be returned (it will be the real runtime) otherwise the default error will be applied on the prediction.

The runtime prediction *Prediction* provided by the *fake* predictor for the job $\alpha$ with runtime $r$ [3] when evaluating the quantitative errors it is computed as:

- $Prediction = r + r * U[0, 0.05]$ (an almost perfect prediction) when it matches the category that is evaluated in the experiment.

- Otherwise, the returned prediction will follows the uniform distribution $U[max(1, r - \frac{PredError*r}{100}), r + \frac{PredError*r}{100}]$ where the $PredError$ follows the normal distribution $PredError \sim N(error, stdev)$.

[3] Note that this runtime $r$ of the job is provided to the fake predictor by the simulator. It has all the job information in the input workload that is being simulated.

| **Workload** | **RT boundaries**(Seconds) | | %jobs |
|---|---|---|---|
| l_lanl_cm5_cln | Lower bound | 224.27 | 50% shorts |
| | Upper bound | 5606 | 20% larges |
| l_sdsc_par95 | Lower bound | 609 | 80% shorts |
| | Upper bound | 15240 | 10% larges |
| l_sdsc_par96 | Lower bound | 621.34 | 65% shorts |
| | Upper bound | 15534 | 15% larges |
| l_kth_sp2 | Lower bound | 956.88 | 55% shorts |
| | Upper bound | 23922 | 15% larges |
| l_ctc_sp2 | Lower bound | 1152.2 | 55% shorts |
| | Upper bound | 28804 | 20% larges |

Table 5.1: Run time boundaries for the workloads (seconds)

| **Workload** | **Processors boundaries** | | %jobs |
|---|---|---|---|
| l_lanl_cm5_cln | Lower bound | 32 | 60% low cpus |
| | Upper bound | 256 | 25% high cpus |
| l_sdsc_par95 | Lower bound | 2 | 30% low cpus |
| | Upper bound | 24 | 20% high cpus |
| l_sdsc_par96 | Lower bound | 8 | 65% low cpus |
| | Upper bound | 32 | 15% high cpus |
| l_kth_sp2 | Lower bound | 4 | 55% low cpus |
| | Upper bound | 16 | 10% high cpus |
| l_ctc_sp2 | Lower bound | 4 | 30% low cpus |
| | Upper bound | 32 | 20% high cpus |

Table 5.2: Number-of-processors boundaries for the workloads

### 5.5.2 Prediction Generation Using Qualitative Errors

In the above subsection we presented how quantitative errors are computed. As will be explained in the *experiments evaluation* section, once the experiments with these kind of errors were analyzed, we realized that although adding an amount of time in the prediction has clear effects in some categories (mainly on the short jobs) we could be more aggressive. We decided to test what would happen to the system performance if, with the job prediction, the nature of the job was changed, for example predicting that a job is short when it is large. This section describes the experiments that we designed to

| **Workload** | **RT boundaries**[4] | | %jobs |
|---|---|---|---|
| l_lanl_cm5_cln | Lower bound<br>Upper bound | 2144<br>5606 | 45% low area<br>24% high area |
| l_sdsc_par95 | Lower bound<br>Upper bound | 1100<br>29724 | 80% low area<br>10% high area |
| l_sdsc_par96 | Lower bound<br>Upper bound | 921<br>311260 | 65% low area<br>15% high area |
| l_kth_sp2 | Lower bound<br>Upper bound | 4321<br>48134 | 55% low area<br>15% high area |
| l_ctc_sp2 | Lower bound<br>Upper bound | 3790<br>53003 | 55% low area<br>20% high area |

Table 5.3: Area boundaries for the workloads (seconds)

test the impact of these kinds of errors.

Qualitative errors model behavior that predictors frequently make which users should not: completely mistaking the job runtime prediction. Usually, users tend to overestimate their jobs by around two or even four times the real runtime of the job. This is because they have an approximate idea about the required time for their applications and they want to avoid their jobs being killed. However, predictors probably will not have the same knowledge concerning the submitted applications and they could make huge mistakes in job runtime predictions. The experiment application categories that have been used for these qualitative errors are:

1. short2large: A percentage of short jobs that will be predicted as large jobs predicting its runtime bigger than the *upper bound* of the *run time* variable.
2. large2short: A percentage of large jobs that will be predicted as short jobs predicting its runtime lower than the *lower bound* of the *run time* variable.
3. short2large and large2short: A percentage of large and short jobs that will be predicted as short jobs and large jobs respectively.

**The prediction:**

The inputs for the experiments when evaluating the impact of qualitative errors on scheduling are:

- The **policy** and **workload** used in the simulation (we used the same policies that were used in the quantitative analysis).
- The **percentage** and **standard deviation** (referenced as *stdev* in the below formula) of jobs to which prediction will imply changing their nature.
- The **conversion** type that indicates which types of jobs are predicted with a runtime that changes their nature.

The runtime prediction *Prediction* provided by the *fake* predictor for the job $\alpha$ with a runtime *r* when evaluating the qualitative errors it is computed as:
Given a uniform variable $\gamma \in U[0..100]$

- $Prediction = r + r * U[0, 0.05]$ if $\gamma > \beta$ where $\beta$ follows a the normal distribution $\beta \sim N(percentage, stdev)$.
- Otherwise:
  - If the job is a small job and the category that is being evaluated is ”**short2large**” or ”**short2large and large2short**” then the prediction is computed by adding the *upper bound* of the *run time* variable to the real job runtime *r*. In this way, the prediction will indicate to the scheduler that the job is large, while, in fact, it is short.
  - If the job is big and the category evaluated is ”**large2short**” or ”**short2large and large2short**” the prediction is done by subtracting the the *upper bound* of the *run time* variable to the job real runtime *r*. In the case that the job prediction is bigger than the lower bound (this can happen with the largest jobs) it is returned with the static value the *lower bound* of the *run time* minus 1.
  - If the two previous conditions are not satisfied then the initial almost perfect prediction is returned.

## 5.6 Experiment Characterization

Our study focuses on a set of four different backfilling scheduling variants: EASY-backfilling, SJBF and LXWF-Backfilling. We used four different workloads: the San-Diego Supercomputer Center 95/96 traces (SDSC), the Cornell Theory Center (CTC) SP2 log, the Swedish Royal Institute of Technology (KTH) SP2 log, and the Los Alamos National Lab (LANL) CM-5 log.

The inputs for the simulations are basically: parameters that allow the choice of policy to be used, what kind of prediction has to be used, parameters to tune the synthetic predictions used in the study and, finally, the workload trace that will be used in the simulation. The trace is based on the standard workload format (SWF) proposed by Feitelson et al. (16) (29)

The experiments consisted of simulating all the different backfilling policies but, instead of using the user estimation provided in the workload, we provided the scheduler with the prediction generated in a fake predictor. The next two subsections present the experiments that were used to carry out the evaluation.

### 5.6.1 Quantitative Errors Experiments

The parameters for the experiments carried out to evaluate the impact of quantitative errors on runtime prediction are presented below:

- **Policies**: EASY-Backfilling, SJBF and LXWF-Backfilling.
- **Errors**: the following errors were used: $\{5, 100, 200, 400, 600, 700, 800, 1000, 10000\}$.
- **Standard deviations**: joint to the last errors two groups of standard deviations were used: $\{0.5, 2, 10, 15, 20, 25, 25, 25, 25\}$ and $\{0.5, 20, 40, 60, 80, 120, 200, 300, 1000\}$.
- **Quantitative Categories**: 1, 2, 3, 4, 5, 6, 7, 8 , 9 and 10.

We tested two different standard deviations because we wanted to see the effect of adding more variation to the prediction (more chaos).

### 5.6.2 Qualitative Errors Experiments

The parameters for the experiments carried out to evaluate the impact of qualitative errors on runtime prediction are presented below:

- **Policies**: EASY-Backfilling, SJBF and LXWF-Backfilling.
- **Percentage of jobs to be converted**: for those policies that use predictions the following percentages have been used $\{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.
- **Standard deviations**: joint the last percentages the two groups of stdevs have been used $\{0.05, 0.1, 0.2, 1, 2, 3, 4, 4, 4, 4, 5, 5\}$.
- **Qualitative Categories**: 1, 2 and 3.

## 5.7 Experiment Evaluation

Three different types of figures are discussed: first the performance for the pure equal category; second a comparison of the quantitative categories, including: the equal, short and large category; and finally the analysis of the qualitative categories is presented. All of them show the $95_{th}$ percentile of the bounded slowdown with the different scheduling policies and workloads.

The evaluation results for the quantitative prediction categories high number of processors, low number of processors, high area, low area, more executed applications and low processors and low runtime did not show relevant impacts on the scheduling performance. Hence, their performance is not shown in the figure analysis. On the other hand, we have stated how the performance impact of the prediction errors shows similar patters in all the presented backfilling variants. Thereby, there are no qualitative differences with the conclusions of our study considering the different evaluated policies.

### 5.7.1 Quantitative Prediction Errors

#### Pure Equal Category

The first interesting fact that can be observed in the figure 5.1 is that the $95_{th}$ percentile of the bounded slowdown for the pure equal does not follow any pattern. The bounded slowdown tendency for the workload kth_sp2 shows higher values with respect to the other two four workloads. This is because 40% of the jobs of this workload are really small (less than one minute), while in the rest of the workloads this percentage goes from 10% to 20%. This clearly shows that when a given workload has an important amount of really short jobs, if the job runtime predictions are inaccurate, this has an incredible effect on the bounded slowdown. From figure 5.1, we can state that there are no clear effects of quantitative errors when they are applied to all the jobs in the bounded slowdown. Even using an error of 10000% the scheduling performance is not influenced. For example in the sdsc_par96 figure with the SJBF and LXWF policies and an error of 5% there is a bounded slowdown of 2 and 2.5 respectively, and there is an bounded slowdown of 1 and 1.4 for the same workloads and policies with an error of 1000%. This behaviour explained by the fact that the pure equal category is adding quantitative errors to all the job prediction. Using this category, the global effect is that the job runtime is over scaled and has no effect on backfilling. Thus, adding a constant error to all the prediction has no collateral effects.

Given these results discussed we defined the **equal category**, where the error is added to 50% of all the jobs (rather than 100% of the jobs used in the **pure equal**

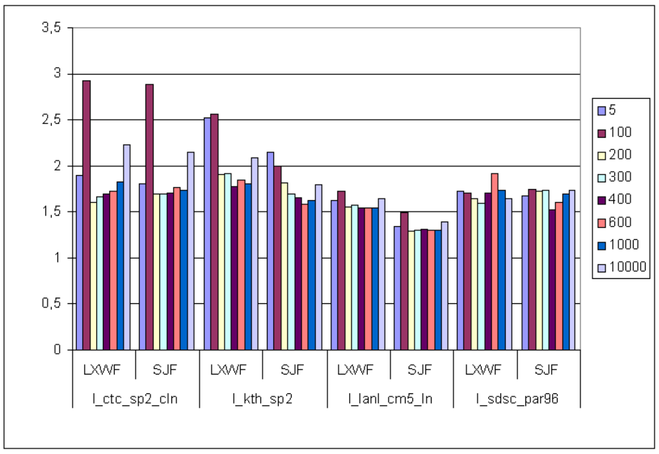


Figure 5.1: BSLD - Pure equal category.

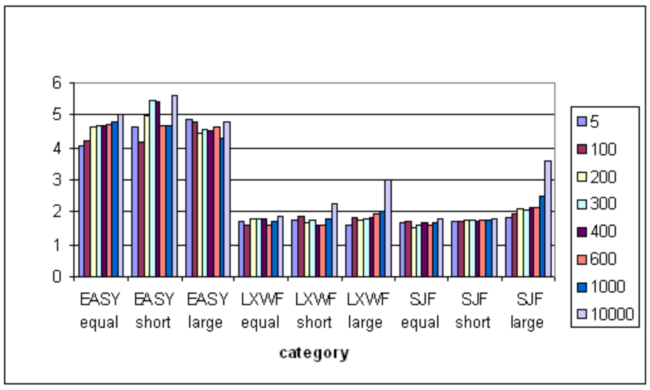


Figure 5.2: BSLD - Quant. Errors - SDSC

category); the **short** category, where the jobs that are short were predicted perfectly and the error was applied to the rest of the jobs; and the **large** categories, where the large jobs were predicted perfectly and the error was applied to the rest of the jobs. The evaluations of the other categories are not presented as they did not have any signifant impact on the system performance.

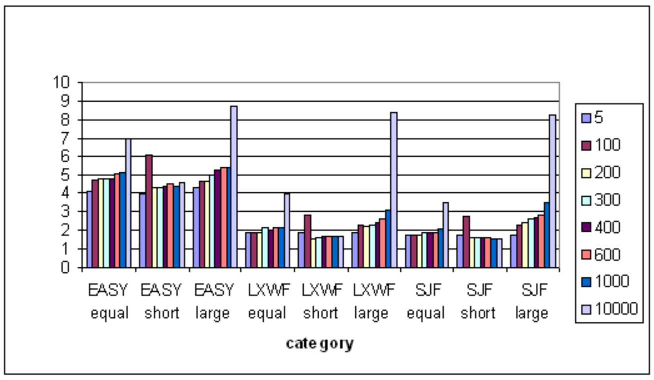


Figure 5.3: BSLD - Quant. Errors - CTC

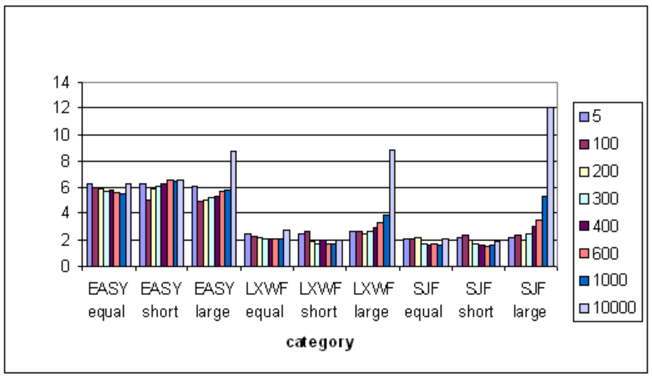


Figure 5.4: BSLD - Quant. Errors - KTH

### Large, Short and Equal Categories

In the equal category, not only is the performance of the system not damaged by the inaccuracy of the prediction errors, but in some situations it actually improves by one order of magnitude with respect to a perfect prediction. For example, the bounded slowdown for SJBF Backfilling policy in the l_kth_sp2 is reduced from 3 with an almost perfect prediction until 2 with an error of 1000%. The inaccuracy of the runtime estimates can lead to very good scheduling, and errors around 200% or even 400% in the runtime prediction can lead to a better schedule than perfect estimations. The impact of this inaccuracy has been called the ”Heel and Toe Effect” by Tsafrir in his studies of the impact of user runtime estimates on backfilling policies (98). However the scheduling

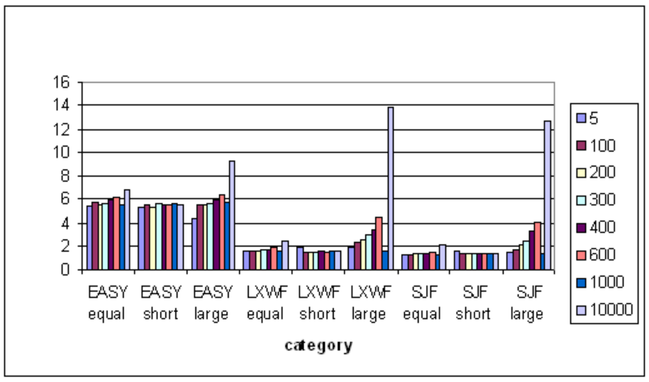


Figure 5.5: BSLD - Quant. Errors - LANL

performance starts to be affected by really high quantitative errors (more than 5000%)

The large category was shown to have more effect on the performance of the system. In the figure 5.3 the bounded slowdown for the ctc_sp2 workload and for all the policies presents a clear effect of the error when the large category is used. The large policy predicts with high accuracy large jobs, which means that the errors are mainly being added to the short jobs. Thus, this study corroborates that prediction accuracy (in this study positive and negative errors) on short jobs has a relevant impact on system performance. On the other hand, the short category, which predicts with accuracy short jobs and adds errors to the rest of jobs, is not highly affected by the increment of the prediction error. The short category shows that there are no clear effects of predictions error on non shorts jobs. The kth_sp2, sdsc_par96, and lanl_cm5 workloads present similar patterns as the ctc2_sp2 workload (see figures 5.2 5.4 and 5.5).

In line with the conclusions that the user runtime estimate impact studies found, we can state that the accurate prediction of short jobs is an important factor, independently of positive or negative errors. Accuracy for large jobs is also important, but their impact on policy performance is not as dramatic as with short jobs, higher errors being acceptable.

In general, we found out that there is no common pattern when adding quantitative errors on job runtime prediction. The only categories that showed an impact on the performance of scheduling policy were the large and the equal categories. However, in this last category the when quantitative errors are really high the performance starts to be affected. For example with the *SDSC* workload, the BSLD grows from 2 with an error of 200% until 5 with an error of 1000%. Prediction accuracy for short jobs is an important factor which can lead to good scheduling. For instance with the *CTC* workload, the

BSLD grows from 2 with an error of 100% until 8 with an error of 10000%. We expected prediction errors in the categories based on the amount of processors (high/low number of processors used and high/low area categories) to have higher impacts on the performance of the system. However, as we have stated, carrying out accurate predictions on these topologies of jobs does not have dramatic effects.

### 5.7.2 Qualitative Prediction Errors

The main goal of our study was to find out what kind of errors have real impact on scheduling performance. Because we realized that there were clear effects with quantitative errors in the categories that were defined based on runtime rather than other variables (such as the number of processors), we decided to take more drastic measures: instead of adding quantitative errors in the categories short and large, we changed the nature of job prediction by adding qualitative errors to job prediction.

This new approach also gave us significant results, and corroborated the results obtained in previous experiments. Figures 5.6, 5.7, 5.8 and 5.9 present the bounded slowdown for the different workloads evaluated in these experiments. There is a clear effect when using the categories **short2large** and **large2short** on the scheduling when incrementing the percentage of jobs whose nature is being changed: the performance of the system is decreased by a factor of at least two. For instance, in the l_ctc_sp2 (figure 5.6) the bounded slowdown goes from 2, predicting as a large jobs a 5% of the shorts jobs, to 20 using a percentage of 80% of the same jobs. In this last example the bounded slowdown is incremented by 4 times.

However, a heavier impact occurs when changes are applied to all the short and large jobs (using the category **”short2large and large2short”**. Obviously, the number of jobs that are changed in this case is bigger that in the other two categories. However, only changing 50% of jobs that belong to this category produces an important effect on the performance of the system, in some cases increasing the bounded slowdown 7 times. Furthermore, in the worst cases, where 90% of large and short jobs are affected by the change, the bounded slowdown can be incremented 10 times.

Qualitative errors in job runtime prediction have been shown to have a relevant impact on the performance of the system. We have demonstrated that incrementing the percentage of jobs affected by this kind of prediction error leads to an important loss of performance which can be appreciated in backfilling scheduling policies. This study provides important information that should be taken into account when designing predictors that will be used in backfilling scheduling policies. These predictors should try to avoid qualitative errors in their predictions, for example as some have been

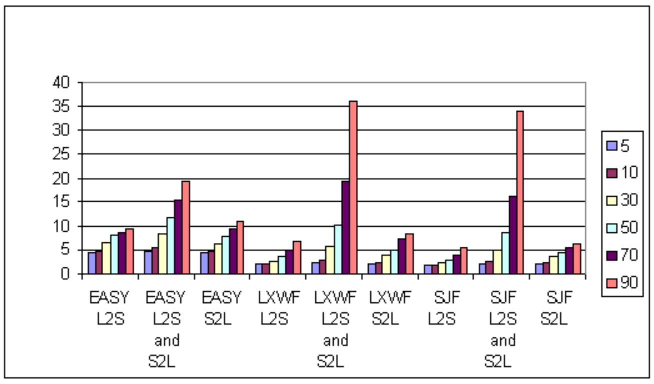


Figure 5.6: BSLD - Qual. Errors - CTC

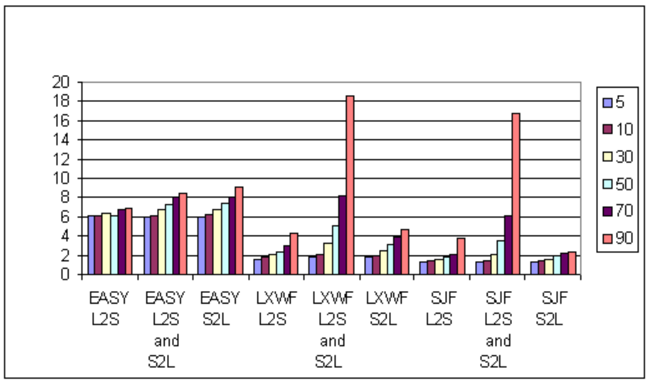


Figure 5.7: BSLD - Qual. Errors - LANL

presented in some studies, providing a confidence interval in the predictions so as to warn that the prediction has either quantitative or qualitative errors.

## 5.8 Summary

We have evaluated of the impact of these prediction models on a set of backfilling variants. The analysis found that adding quantitative errors to the runtime of all the jobs of the workload has no clear impact on the performance of the system, even with prediction errors of 10000%. However, when the errors are applied randomly to 50% of all the stream of jobs, the scheduling performance is clearly affected by high quantitative

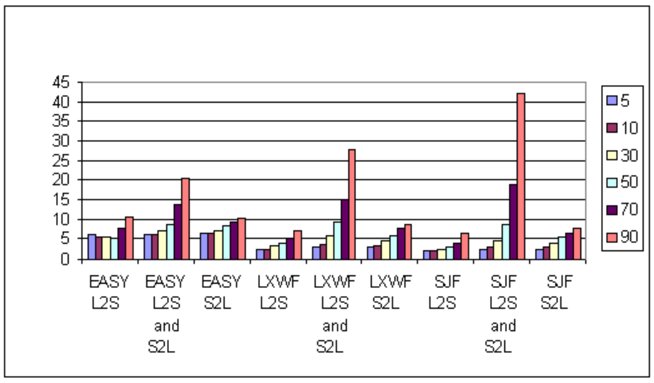


Figure 5.8: BSLD - Qual. Errors - KTH

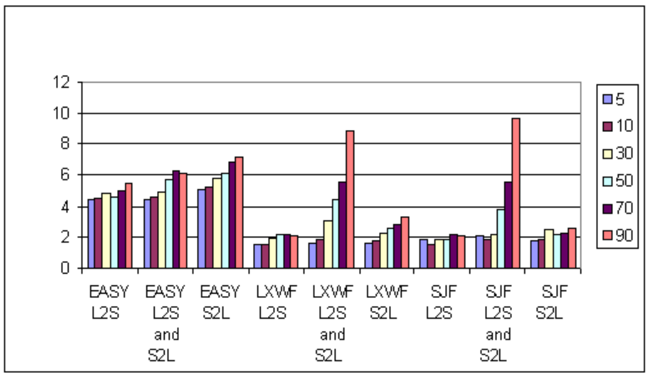


Figure 5.9: BSLD - Qual. Errors - SDSC

errors (more than 5000%). Surprisingly, quantitative negative errors did not have any important impact on the performance of the system as we had expected. However, as it is shown in the qualitative experiments, they only provide relevant impacts when the negative errors are really large.

If the quantitative errors are applied to a determined subset of jobs based on the required resources (lower/high number of processors, lower/high memory used or lower/high $processors * runtime$ area consumed) there is no relevant impact on the performance of the system. However, when these quantitative prediction errors are applied to the prediction of short jobs, the performance of the average bounded slowdown steadily increases for errors of up to 400%, and from this point it experiments an exponential increment. Taking into account the results obtained in this study, we confirm that it is not necessary to design specialized predictors for specific job types based on the resources required to achieve good performance. However, predictors should be as accurate as possible for those jobs that are likely to be short, and try to avoid high quantitative errors in them.

Clearly, qualitative errors have to be avoided when predicting job runtime for short and large jobs. These errors have a clear exponential tendency on the average bounded slowdown for all evaluated workloads when the number of jobs being mis-categorized is incremented.

# Chapter 6

# Modeling the impact of resource sharing in Backfilling Policies

Job scheduling policies for HPC centers have been extensively studied in the last few years, specially backfilling based policies. Almost all of these studies have been done using simulation tools. These tools evaluate the performance of scheduling policies using the workloads and the resource definition as an input.

To the best of our knowledge, all the existent simulators use the runtime (either estimated or real) provided in the workload as a basis of their simulations. However, the runtime of a job, even executed with a fixed number of processors, depends on runtime issues such as the specific resource selection policy used to allocate the jobs.

In this chapter we analyze the impact on system performance of considering the resource sharing of running jobs. For this purpose we have included in the Alvio simulator a performance model that estimates the penalty introduced in the application runtime when sharing the memory bandwidth. Experiments have been conducted with two resource selection policies and we present both the impact from the point of view of global performance metrics, such as average slowdown, and per job impact such as percentage of penalized runtime.

## 6.1 Introduction

Several works focused on analyzing job scheduling policies have been presented in the last decades. The goal was to evaluate the performance of these policies with specific workloads in HPC centers. A special effort has been devoted to evaluating backfilling-based policies because they have demonstrated an ability to reach the best performance results (i.e: (98) or (91)). Almost all of these studies have been done using simulation tools. To the best of our knowledge, all the existent simulators use the runtime (either estimated or real) provided in the workload as a basis of their simulations. However, the runtime of a job depends on runtime issues such as the specific resource selection policy used or the resource jobs requirements.

This chapter evaluates the impact of considering the penalty introduced in the job runtime due to resource sharing (such as the memory bandwidth) in system performance metrics, such as the average bounded slowdown or the average wait time, in the backfilling policies in cluster architectures. To achieve this, we have developed a job scheduler simulator (Alvio simulator) that, in addition to traditional features, implements a job runtime model and resource model that try to estimate the penalty introduced in the job runtime when sharing resources. In this work we have only considered in the model the penalty introduced when sharing the memory bandwidth of a computational node.

We simulated the impact of memory bandwidth usage in the scheduling policy using two workloads trace files (CTC and SDSC) obtained from the Feitelson Parallel Workload Archive. As the original traces did not contain the job memory bandwidth usage, we created it synthetically. For each workload, three scenarios were defined: one with 80% of jobs having a high memory bandwith usage (HIGH), one with 50% of jobs having a medium memory bandwith usage (MED), and one with 10% of jobs having a high memory bandwith usage (LOW). These scenarios were evaluated with the SJF-Backfilling policy, with two resource selection policies (First fit and First Contiguos fit), and with and without considering the penalty of the resource usage sharing.

Results showed a clear impact of system performance metrics such as the average bounded slowdown or the average wait time. Furthermore, other interesting collateral effects such as a significant increment in the number of killed jobs appeared. Moreover, as we describe below, the impact on these performance metrics is not only quantitative.

The rest of the chapter is organized as follows: in section 6.2 we provide the formalizations for model that is used in the simulations; in section 6.3 the experiments that we have evaluated are presented; in section 6.4 the experiments evaluations are presented; and finally in section 6.5 the conclusions and future work are presented.

## 6.2 Modeling Runtime penalty in Alvio

The job scheduling policy uses as input a set of job queues and a Reservation Table (RT). The RT represents the status of the system at a given moment and is linked to the architecture (see figure 6.2). Figure 6.1 presents the cluster architecture model that has been evaluated in this Thesis. The reservation table is a bi dimensional table where the

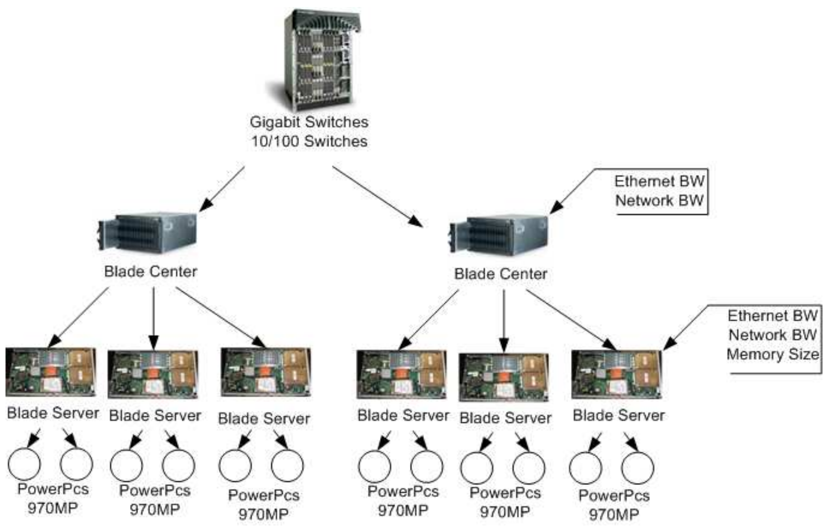


Figure 6.1: Resource Model

$X$ axes represent the time and the $Y$ axes represent the different processors and nodes of the architecture. It has the running jobs allocated to the different processors during the time. One allocation is composed of a set of buckets that indicate that a given job $\alpha$ is using the processors $\{p_0, .., p_k\}$ from *start time* until *end time*. Depending on the experiment configuration, the scheduling policy will allocate temporarily the queued jobs (for instance, to estimate the wait time for the jobs).

In this section we provide a characterization for the different elements managed by the Alvio simulator and for other parts of the evaluated model. First we formalize the entities that are evolved in the scheduling algorithm: the job and the available resources (defined in the RT). Second, we formalize the main characteristics of a scheduling policy and the different elements that it uses for computing the schedule.

### 6.2.1 The Scheduling entities

The main entities that are involved in the scheduling architecture are:

- The *Workload* that contains all the stream of jobs that are submitted to the system.
- The Job $\alpha$ that is submitted to the system is characterized by [1]:

[1]We use the SWF plus an extension to include the resource usage in terms of memory bandwidth

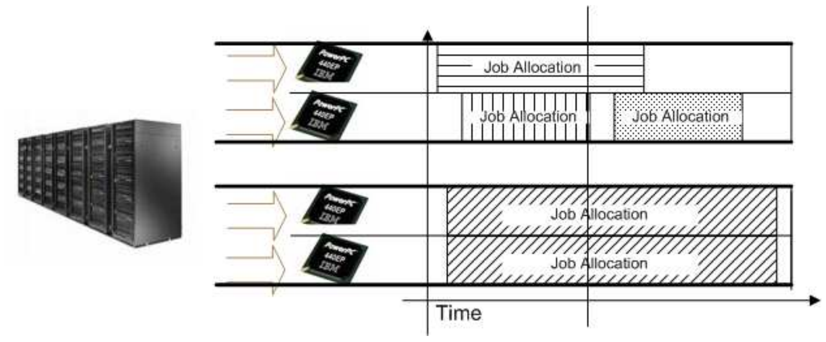


Figure 6.2: The reservation table

- The static description of the job: $description_{\alpha} = \left\{\alpha_{\{JobPropierty_1,value\}},..,\alpha_{\{JobPropierty_n,value\}}\right\}$. In the model presented in this chapter we consider: the job number, user and group identifiers, the executable, the submit time and the runtime.
- The requirements for the job: $requirements_{\alpha} = \left\{\alpha_{\{JobRequirement_1,value\}},..,\alpha_{\{JobRequirement_n,value\}}\right\}$. In the model presented in this chapter we consider: the number of requested processors (referenced as *CPUS*) and the memory bandwidth (Its units are in MB/S). However, future extensions will include the memory required, the ethernet bandwidth and the network bandwidth.

- A set of computational resources $\{\sigma_1,..,\sigma_n\}$ available on the system. An example of computational resource is the typical computing machine. In the evaluations presented in this Thesis we consider that the system is composed by one computational resource. Each resource $\sigma$ is described by:
  - A set of capabilities: $capabilities_{\sigma} = \left\{\partial_{\{1\}},..,\partial_{\{n\}}\right\}$, where each capability is composed by a pair value: $\partial_{\{i\}} = \{ResCapability, value\}$. In this Thesis we have considered that a computational resource has the following capabilities: total number of processors and total number of nodes. Note that they provide general information about the properties of the machine.
  - The computational resource is composed by a set of homogeneous nodes $\{n_1,..,n_k\}$. Each node $n$ of this resource $\sigma$:
    - $*$ is composed by a set of processors $\left\{p_{\{1,n\}},..,p_{\{j,n\}}\right\}$

∗ has a set of shared resources with a certain capacity: $Resources_{\sigma,n} = \left\{r_{\{1,\sigma,n\}},..,r_{\{k,\sigma,n\}}\right\}$. In this Thesis we have considered that each node has only the memory bandwidth as a physical resource with a capacity of 6 GB/Sec [2]. Thereby, the resource list of all the nodes is modeled as: $Resources_{\{n,\sigma\}} = \{\{MemoryBandwidth, 6000MB/S\}\})$. Future extensions of this model will consider also the network bandwidth, ethernet bandwidth and the memory as a physical resources of the nodes.

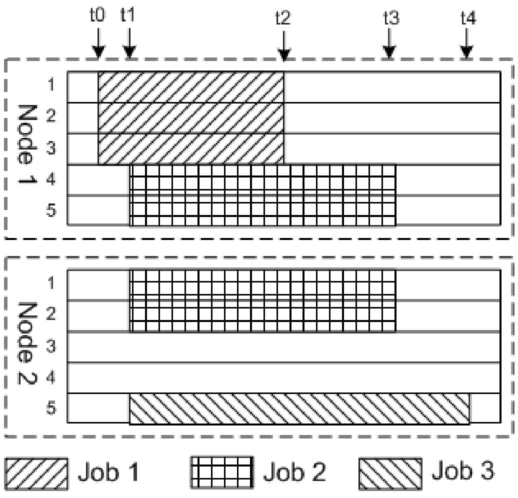


Figure 6.3: Reservation Table Snapshot

### 6.2.2 The Resource Selection Policy

The Resource Selection Policy (RSP), given a set of free processors and a job $\alpha$ with a set of requirements, decides to which processors the job will be allocated. To carry out this selection, the RSP uses the Reservation Table. The Reservation Table contains the mapping for all the jobs that are running in the processors that are currently used. For each job it contains in which processors are its processes allocated and when.

An allocation is defined by: $allocation\{\alpha\} = \left\{[t_0, t_1], P = \left\{p_{\{g,n_h\}}, ..p_{\{s,n_t\}}\right\}\right\}$ [3] and indicates that the job $\alpha$ is allocated to the processors $P$ from the time $t_0$ until $t_1$. The

[2]The configurations evaluated in this Thesis are presented in the Evaluation section

[3]The processor $p_{\{g,n_h\}}$ refers to a processor of the node $n_h$ with local identifier $g$

allocations of the same processors must satisfy that they are not overlapped during the time.

Figure 6.3 provides an example of a possible snapshot of the reservation table at the point of time $t_1$. Currently, there are three jobs running in three different job allocations:
$a_1 = \left\{[t_0, t_2], \left\{p_{\{1,node_1\}}, p_{\{2,node_1\}}, p_{\{3,node_1\}}\right\}\right\}$
$a_2 = \left\{[t_1, t_3], \left\{p_{\{4,node_1\}}, p_{\{5,node_1\}}, p_{\{1,node_2\}}, p_{\{2,node_2\}}\right\}\right\}$
$a_3 = \left\{[t_1, t_4], \left\{p_{\{5,node_2\}}\right\}\right\}$

## The First Fit and First Continuous Fit Resource Selection Policies

In this chapter we describe the impact of resource sharing usage on the Shortest Job Backfilled first scheduling policy using the resources selection policies of First Fit (FF) and First Continuous Fit (FCF). These two selection policies are the simplest policies that can be used by the local resource manager.

The FF policy, given a required start time $t_{req}$ and given the job $\alpha$ with a requested runtime of $\alpha_{\{RunTime,rt\}}$ and number of requested processors $\alpha_{\{CPUS,p\}}$, finds in the reservation table the first $\alpha_{\{CPUS,p\}}$ processor allocations whose start time is the closest to the provided time and whose length is higher or equal to $\alpha_{\{RunTime,rt\}}$. To do this, the selection policy follows these steps:

1. For each processor $p_i$ [4] in the reservation table, the resource selection policy selects the interval of time $[t_{x_i}, t_{y_i}]$ that is closest to the $t_{req}$ that satisfies it: no process is allocated to the processor during the interval, and its length is equal to or greater than $\alpha_{\{RunTime,rt\}}$. We define this interval of time as a *bucket*. Thus the $b_{(i,t_{i_0},t_{i_1})}$ bucket is defined as the interval of time $[t_x, t_y]$ associated to the processor $p_i$. All the selected intervals of time are added to the set *Buckets* where they are strictly ordered by the interval start time $t_{i_0}$. In the example provided in the figure 6.4, the returned set would be $\{b1, b2, b7, b8, b9, b3-6, b13\}$

2. Given all the different buckets $\left\{b_{(1,t_{1_0},t_{1_1})}, .., b_{(N,t_{N_0},t_{N_1})}\right\}$ associated to the reservation table that are included in the ordered set *Buckets* (where $N$ is the number of processors available on the reservation table), the resource selection policy FF will select the first $\alpha_{\{CPUS,p\}}$ buckets that satisfy the condition that their interval of time shares at least the runtime required by the job. For instance, in the

[4] For the description of the resource selection policy we refer to the processor $p$ with a subscript $i$ that indicates the global identifier of the processor in the computational resource (from 1 until $N$). Thus the processors $p_1$ refers to the processor with global identifier 1.

figure 6.4 the FF-RSP will select the buckets *b1*, *b2* and *b7* to allocate a job whose runtime is $\alpha_{\{RunTime,30secs\}}$ and requests $\alpha_{\{CPUS,3\}}$ processors.

3. If the preceding step the RSP could select the required number of buckets, the allocation that is returned is $allocation\{\alpha\} = \left\{\left[t_0, t_0 + \alpha_{\{RunTime,rt\}}\right], P\right\}$ where the set of number of processors $P$ is based on the information included in the selected buckets, and the $t_0$ is computed as the minimum start time that satisfies that all the intervals of times of the selected buckets are included in $\left[t_0, t_0 + \alpha_{\{RunTime,rt\}}\right]$.
4. In the case that in step *2* the number of buckets that satisfied the required conditions was lower than the required processors, there were not enough buckets that shared the required amount of time. In this situations, the bucket with start time greater than the head of the set of buckets defined in the step 1 $b_{(i,t_{j_0},t_{j_1})}$ and with start time greater to $t_{req}$ is selected, the $t_{req}$ is updated as $t_{req} = t_{j_0}$ and the steps *1*, *2* and *3* are iterated again until a valid allocation is found.

The figure 6.5 exemplifies the situation described in the last step: if a job with $\alpha_{\{RunTime,20secs\}}$ and $\alpha_{\{CPUS,6\}}$ is required to be allocated, in step *1* the buckets introduced in the previous example would be selected $\{b1, b2, b7, b8, b9, b3-6, b13\}$. However none of the possible subgroups of six buckets share the amount of required time. Thus, the $t_{req}$ is updated to $t_{req} = t_0$, that is the start time of the bucket *b8*. In this iteration the selected nodes will be exactly the same as in the previous iteration. Thus, no valid allocation will be found. In the next iteration, the $t_{req}$ is updated to $t_{req} = t_1$, that is the start time of the bucket *b4*. In this iteration the set of buckets resulting from step *1* is the following: $\{b1-6, b10-13\}$. In this iteration the algorithm reaches the buckets that satisfy the requirements of the job and the resource selection policy: *b1*, *b2*, *b3*, *b4*, *b5* and *b6*. The returned allocation will be composed by the processors $P = 1;2;3;4;5;6$ and start time of the bucket *b6*.

The First Continuous Fit resource selection policy follows almost the same algorithm that we have described in the previous paragraph. The steps *1*, *2*, *3* and *4* are also followed by the selection algorithm. However, a new restriction is added to the step *2*. The selected buckets not only have to satisfy the condition that the interval of time is at least equal to the runtime required by the job, they also have to satisfy the condition that the buckets are consigned to consecutive number of processors. Thus in the first example, the RSP could not select the buckets *Bucket 1*, *Bucket 2* and *Bucket 7*. It should select the buckets *Bucket 7*, *Bucket 8* and *Bucket 9*. Note that using this resource selection policy the interval of time returned in the allocation would be substantially delayed with respect to that obtained by the First Fit resource selection policy.

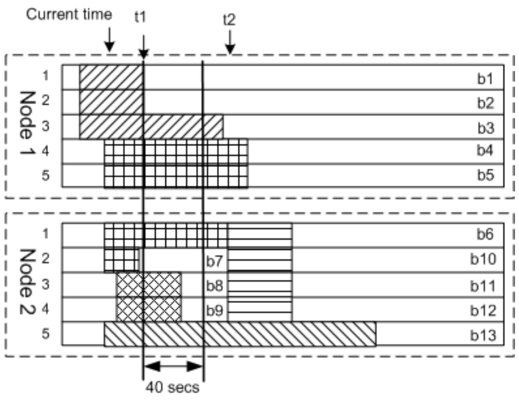


Figure 6.4: Fist Fit Example 1

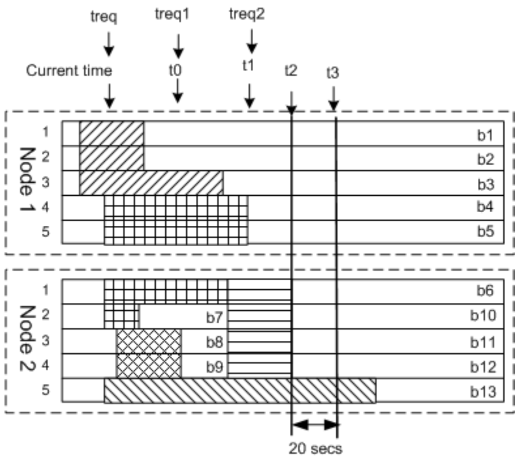


Figure 6.5: First Fit Example 1

### 6.2.3 The Job Scheduling Policy

The other component of the scheduling architecture that has to be considered in the modelization is the scheduling policy. It is the component that decides which of the queued jobs should start based on the information and functionalities provided by the local resource manager that implements a given resource selection policy (see figure 6.6). The scheduling policy is characterized by:

- The Job Scheduling Algorithm, that given a set of jobs and the information provided by the Resource Selection Policy, decides which job to start. The algorithm is applied taking into account the different allocations that the queued jobs would obtain based on the resource selection policy that is used. Thus, the local scheduler requires the estimation of the start time of a given job providing its estimated runtime $\alpha_{\{EstimatedRunTime,rt\}}$ and its requested processors $\alpha_{\{CPUS,3\}}$.

- The Backfilling Queue that contains the set of jobs that are currently considered when carrying out the backfilling process.
- The wait queue that contains the set of jobs that are currently waiting for be executed. Usually, the jobs are queued following a given criteria. The most commonly used is the First Come First Server. However, as has been introduced in the previous chapters, other sorting criterion may be used (Shortest Job First, LXWF etc.)

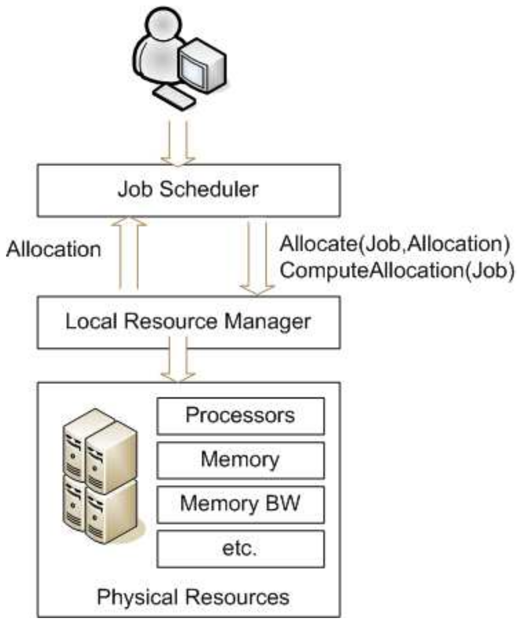


Figure 6.6: Local Resource and Scheduler interaction

### 6.2.4 Modeling the conflicts

The model that we have presented in the previous section has some properties that allows us to simulate the behavior of a computational center with more details. Different resource selection policies can be modeled. Thanks to the Reservation Table, it knows at each moment which processors are used and which are free. Thus, while simpler models do not do this, our model allows us to select specific processors for a given job (different resource selection policies). For example: a first continuous policy, that allocates consecutive processors to the jobs can be simulated.

Using the resource requirements for all the allocated jobs, the resource usage for the different resources available on the system is computed. Thus, using the Reservation Table, we are able to compute, at any point of time, the amount of resources that are being requested in each node. Thus, if in a interval of time $[t_0,t_1]$ the resource capacity $\partial$[5] of a node $n$ used by the allocated processes is higher than the capacity of the resource a penalty to of time proportionally to the resource consumption and the time $t_1 - t_0$ is added to each process.

In this extended model, when a job $\alpha$ is allocated during the interval of time $[t_x,t_y]$ to the reservation table to the processors $p_1,..,p_k$ that belong to the nodes $n_1,..,n_j$, we check if any of the resources that belong to each node is overloaded during any part of the interval. In the affirmative case a runtime penalty will be added to the jobs that belong to the overloaded subintervals.

Before introducing the model used to calculate the penalties, we formalize some concepts used later: The *Shared Windows* and the *Penalty Function*.

#### The Shared Windows

A *Shared Window* is an interval of time $[t_x,t_y]$ associated to the node $n$ where all the processors of the node satisfy the condition that: either no process is allocated to the processor, or the given interval is fully included in a process that is running in the processor.

Figure 6.7 shows the Shared Windows that would be defined in the example introduced in the figure 6.3. In this example six different Shared Windows would be defined, four in the first node and two in the second node. For instance: the Window 5 is defined in the interval $[t_1,t_3]$ in the *node 2* and contains the *job 2* in the processors 1 and 2, the *job 3* in the processor 5 and the processors 3 and 4 remain idle.

[5]By omission, when no subscripts are specified we refer to a non specific element. For instance if we are describing a property of the resource $\partial$, we are concern to a given resource of a given node and computational resource but without specifying which.

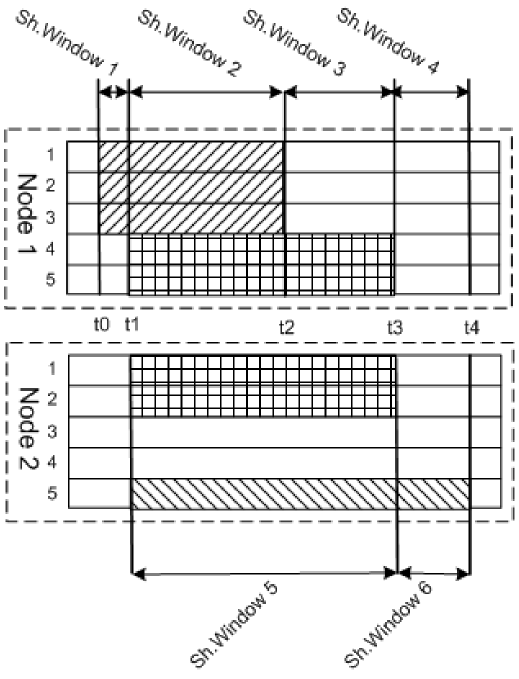


Figure 6.7: Example of Shared Windows definition

### The penalty function

This function is used to compute the penalty that is associated with all the jobs included to a given Shared Window due to resources overload. The input parameters for the function are:

- The interval associated to the Shared Window $[t_x, t_y]$.
- The jobs associated to the Shared Window$\{\alpha_0, .., \alpha_n\}$
- The node *n* associated to the Shared Window with its physical resources capacity.

The function used in this model is defined as [6]:

[6]Note that all the penalty, resources, resource demands and capacities shown in the formula refer to the node *n* and the interval of time $[t_x, t_y]$. Thereby, they are not specified in the formula

$$\forall res \in resources(n) \rightarrow demand_{res} = \sum_{\alpha}^{\{\alpha_0,..,\alpha_n\}} r_{\alpha,res} \tag{6.1}$$

$$Penalty = \sum_{resources(n)}^{res} \left(\frac{\max\left(demand_{res}, capacity_{res}\right)}{capacity_{res}} - 1\right) \tag{6.2}$$

$$PenlizedTime = (t_y - t_x) * Penalty \tag{6.3}$$

First for each resource in the node the resource usage for all the jobs is computed. Second, the penalty for each resource consumption is computed. This is a linear function that depends on the overload of the used resource. Thus if the amount of required resource is lower than the capacity the penalty will be zero, otherwise the penalty added is proportional to the fraction of demand and availability. Finally, the penalized time is computed by multiplying the length of the Shared Window and the penalty. This penalized time is the amount of time that will be added the node penalized time to all the jobs that belong to the Window. This model has been designed for the memory bandwidth shared resource and can be applicable to shared resources that behave similar. However, for other typology of shared resources, such as the network bandwidth, this model is not applicable. Future work will be focused on modelizing the penalty model for the rest of shared resources of the HPC local scenarios that can impact in the performance of the system.

### 6.2.5 Computing the Reservation Table Penalties

In the previous subsection 6.2.4 we introduced the Shared Windows and explained how the penalties for these windows are computed to all the jobs that are allocated to a given set of nodes. In this subsection we explain how the job penalized time is computed. Figure 6.8 shows the penalty associated with each Window.

The algorithm is applied to the reservation table with all the allocated jobs and with no penalties assigned. It has to be applied each time that a job is allocated in the reservation table and it is divided in three main steps:

1. The shared windows for all the nodes and the penalized times associated with each of them are computed. In the example, only Windows 2 (with penalty of *p1*) and 5 (with penalty of *p2*) have associated penalties. This means that during the intervals of both Windows the resource demand for the allocated jobs was higher than the available capacity for the resources.

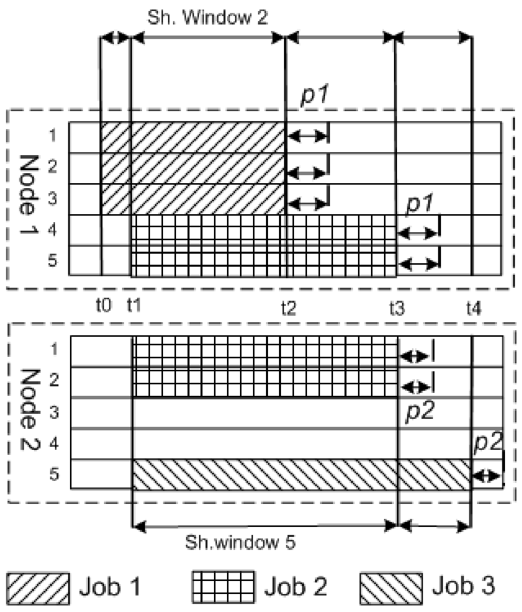


Figure 6.8: Example of Shared Window Penalties

2. The penalties of each job associated with each node are computed adding the penalties associated with all the windows where the job runtime is included. For example, as can be observed in the figure 6.9 , the penalty for the *job 2* in the *node 1* will be *p1*. In the case that the window 1 would have a penalty of *p3*, the penalty of *job 1* would have been $p3 + p1$.
   Thus, if the job $\alpha$ is allocated to the the nodes $n_1,..,n_k$, it will have $k$ different amount of penalized times ($\{ptime_1,..,ptime_k\}$).

3. The penalty time that will be finally assigned to the job will be the maximum of the penalties that job would have in the different nodes. Figure 6.9 shows how the penalty is assigned to each job. The penalty for the *Job 2* is *p1* due to the penalty that the job suffers for its allocation to node one (*p1*) is bigger than the penalty that the job will suffer for be allocated to node two (*p2*). The finalization time for all the jobs is updated in the Reservation Table according to the computed penalties.

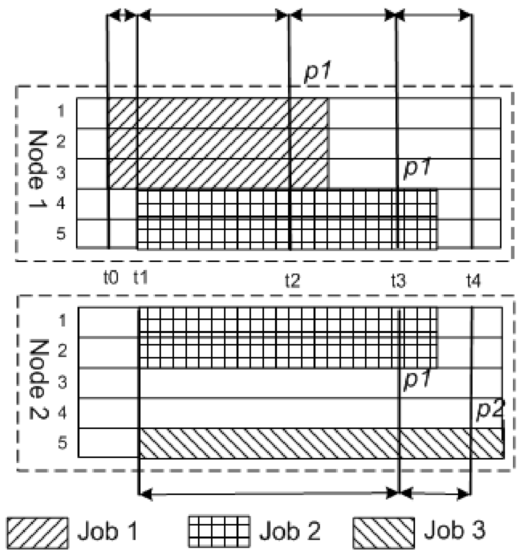


Figure 6.9: Example of Final Job Penalties

## 6.3 Experiments

In this chapter we evaluate the effect of considering the memory bandwidth usage when simulating the Shortest Job Backfilled Firts policy under several workloads and resource selection policies. Two different workloads from the Feitelson workload archive (29) have been used. For each of them we have generated three different scenarios: with high (HIGH), medium (MED), and low (LOW) percentage of jobs with high memory demand.

### 6.3.1 Workloads

For the experiments we used the cleaned (99) versions of the workloads SDSC Blue Horizon (SDSC-BLUE) and Cornell Theory Center (CTC) SP2. For the evaluation experiments explained in the following section, we used the first 10000 jobs of each workload. Based on these workload trace files, we generated three variations for each one with different memory bandwidth pressure:

- HIGH: 80% of jobs have high memory bandwidth demand, 10% with medium demand and 10% of low demand.
- MED: 50% of jobs have high memory bandwidth demand, 10% with medium

demand and 40% of low demand.

- LOW: 10% of jobs have high memory bandwidth demand, 10% with medium demand and 80% of low demand.

### 6.3.2 Architecture

For each of the workloads used in the experiments we defined an architecture with nodes of four processors, 6000 MB/Second of memory bandwidth, 256 MB/Second of Network bandwidth and 16 GB of memory. In addition to the SWF traces with the job definitions we extended the standard workload format to specify the resource requirements for each of the jobs. Currently, for each job we can specify the average memory bandwidth required (other attributes can be specified but are not considered in this work).

Based on our experience and the architecture configuration described above, we defined that a *low memory bandwidth demand* consumes 500 MB/Second per process; a *medium* memory bandwidth demand consumes 1000 MB/Second per process; and that a *high memory bandwidth demand* consumes 2000 MB/Second per process.

Based on the job runtime model used, the maximum penalty in the job runtime for a given window will take place in those cases that the amount of memory bandwidth requested per processors is 2000 MB/Sec. Thus the penalty would be $\frac{8}{6}$.

### 6.3.3 Experiments

For each of the different scenarios, the following configurations were evaluated:

1. Shortest Job Backfilled First Backfilling scheduling policy.
2. Two different Resources Selection Policies (RSP) have been evaluated: **First Fit** (FF:jobs processes are allocated to the first available set of processor); **First Continuous Fit** (FC:jobs processes are allocated to the first set of continuous processors).
3. Using as execution time the runtime existing in the workload trace file (traditional modelization) and using our job runtime model (with resource sharing modelization).

| **Center** | **RSP** | **B-SLD** | **Wait time** | Killed Jobs |
|---|---|---|---|---|
| | FF | 1.9 | 1390 | 30 |
| CTC | FC | 1.7 | 3890 | 30 |
| | FF | 32 | 31008 | 2 |
| SDSC | FC | 64 | 44301 | 2 |

Table 6.1: System performance metrics using runtime=trace file runtime - $95_{th}$ Percentile

| **Center** | **RSP** | **B-SLD** | **Wait time** | Killed Jobs |
|---|---|---|---|---|
| | FF | 2,6 | 2860 | 30 |
| CTC | FC | 2,1 | 5530 | 30 |
| | FF | 4,2 | 2968 | 2 |
| SDSC | FC | 15,4 | 6351 | 2 |

Table 6.2: System performance metrics using runtime=trace file runtime - Average

## 6.4 Evaluation

Table 6.1 shows the $95_{th}Percentile$ for the bounded slowdown (B-SLD), the $95_{th}Percentile$ for the wait time (WT) and the number of killed jobs (KJ) [7] for the simulations of the workloads using the trace file provided job runtime for each of the Resource Selection Policies (RSP). Table 6.2 shows the same information but with the average. Clearly, the performance of both workloads depends on the resource selection policy used. Surprisingly, for the CTC workload the First Continuous policy (FC) achieves better performance than the First Fit policy(FF). Analyzing the simulation in detail, we stated that in a very loaded interval of time, some jobs that require a high number of processors are delayed in the FC due to no continuous processors being available. Thus all the short jobs are backfilled and the system becomes less loaded more quickly. On the other hand, with the FF policy, these jobs start before and all these short jobs are substantially delayed. Thus, the load of the system decreases more slowly than with the FC. The SDSC workload shows the behavior that *could* be considered more normal. In this case the First-Fit policy achieves the best performance in terms of wait time and bounded slowdown.

Tables 6.3 and 6.4 show the System performance metrics using the $95_{th}$ Percentile and the average respectively when executing, using as job runtime model the mode

[7]Kills due to the runtime was higher than the estimation provided by the user

proposed in this work that includes the penalty estimation based on the conflicts generated by resource sharing. In addition to the information shown in Table 6.1, we also show the average of the percentage of penalized runtime (%PRT).

The results show quantitative differences in the performance obtained when using the trace file runtime. In both workloads, incrementing the number of jobs with high memory bandwidth demand resulted in an increment of all the performance metrics. In CTC evaluations, the performance metrics showed a clear increment in all the experiments. For instance, in the LOW scenario for the CTC with FF, the B-SLD grew from 1,9 (from the original model) to 2,2 (a 15% increment). Furthermore, the BSLD in the HIGH scenario for the CTC with FF grew from 1.9 to 4.2 (a 120% increment). Similar effects can be appreciated in the SDSC workload. For example, using the FC policy, the wait time $95_{th}Percentile$ grew from the 44301 sec. of the traditional modelization to 44940 in LOW scenario, and to 96803 in HIGH scenario.

Another important effect of the resource sharing model concerns the number of killed jobs. For instance they grow from 30 to a maximum of 527 jobs in the worst case for the CTC (HIGH scenario and FC policy), and from 2 to 548 in the worst case for the SDSC (HIGH scenario and FC policy). These numbers are also consistent with the fact that the percentage of the runtime penalty introduced when using a FC policy is higher than when using a FF policy. This conclusion seems to be reasonable because the probability that a job generates an *internal* conflict is higher when processors are consecutive.

Qualitative differences also appeared in the results. The bounded slowdown, for the CTC-SP2 workload using the workload with a medium percentage of jobs with high memory demand concludes that the FC resource selection policy gives better performance than the FF resource selection policy. However, using the traditional model the bounded slowdown results state that the FF policy performance is better than the FC policy.

## 6.5 Summary

In this chapter we have evaluated the impact of considering the penalty introduced in the job runtime due to resource sharing (such as the memory bandwidth) in system performance metrics, such as the average bounded slowdown or the average wait time, in the backfilling policies. This evaluation has been done using simulation techniques with the Alvio simulator. In this chapter we have described the different components and elements that are used in its simulation model. We have formalized the model that characterize how the resources are being shared by the allocated jobs. This model punishes the run time for the running jobs that are sharing a given resource in those cases

| **Center** | **MT** | **RSP** | **B-SLD** | **WT** | **% PRT** | **KJ** |
|---|---|---|---|---|---|---|
| CTC | High | FF | 4.2 | 10286 | 8.8 | 428 |
| | Med | FF | 2.8 | 8963 | 4.8 | 247 |
| | Low | FF | 2.2 | 4898 | 0.92 | 64 |
| | High | FC | 3.5 | 4527 | 11.1 | 527 |
| | Med | FC | 3.1 | 3755 | 6.17 | 333 |
| | Low | FC | 2.1 | 2679 | 1.2 | 80 |
| SDSC | High | FF | 99.3 | 55667 | 11.8 | 475 |
| | Med | FF | 55.4 | 44346 | 6.7 | 255 |
| | Low | FF | 37.7 | 32730 | 1.5 | 51 |
| | High | FC | 197 | 96803 | 13.8 | 548 |
| | Med | FC | 149 | 62883 | 7.5 | 307 |
| | Low | FC | 85.9 | 44940 | 1.4 | 66 |

Table 6.3: System performance metrics using runtime=job runtime model - $95_{th}$ Percentile

| **Center** | **MT** | **RSP** | **B-SLD** | **WT** | **% PRT** | **KJ** |
|---|---|---|---|---|---|---|
| CTC | High | FF | 8,2 | 20082 | 7,8 | 428 |
| | Med | FF | 5,3 | 13443 | 3,8 | 247 |
| | Low | FF | 3,2 | 7124 | 0,72 | 64 |
| | High | FC | 7.1 | 8732 | 9.01 | 527 |
| | Med | FC | 4.02 | 5152 | 7.71 | 333 |
| | Low | FC | 2.59 | 3490 | 1.41 | 80 |
| SDSC | High | FF | 22,56 | 55667 | 15,1 | 475 |
| | Med | FF | 11,32 | 44346 | 10,2 | 255 |
| | Low | FF | 7,54 | 32730 | 5,2 | 51 |
| | High | FC | 197 | 96803 | 12.72 | 548 |
| | Med | FC | 149 | 62883 | 5.452 | 307 |
| | Low | FC | 85.9 | 44940 | 1.24 | 66 |

Table 6.4: System performance metrics using runtime=job runtime model - Average

that the resource demand is higher than the resource capacity.

In this chapter we evaluate the effect of considering the memory bandwidth usage when simulating the Shortest Job Backfilled First policy under several workloads and resource selection policies. Two different workloads have been used in the experiments. For each of them we have generated three different scenarios: with high (HIGH), medium (MED), and low (LOW) percentage of jobs with high memory demand.

The results have shown quantitative differences between the performance obtained using the presented model and not. In both workloads, incrementing the number of jobs with high memory bandwidth demand has resulted in an increment of all the performance metrics. In CTC evaluations, the performance metrics have shown a clear increment in all the experiments. Furthermore, the number of killed jobs has experimented an evident increment.

# Chapter 7

# New Scheduling Strategies for Improving Use of Resources

In this chapter, using the results obtained by the use of the new resource usage model presented in the previous chapter, we present two new different resource selection policies (the **Find Less Consume Distribution** and the **Find Less Consume Threshold Distribution**) that are designed to minimize the job runtime penalty caused by the saturation of resources that are shared in computational resources. Furthermore, we provide a new backfilling-based Job Scheduling strategy (**Resource Usage Aware Backfilling**) that cooperates with the local resource manager whose objective is also to minimize the saturation of shared resources.

## 7.1 Introduction

In the previous chapter we introduced a new simulation model that considers how shared resources are used by jobs that are allocated to the same nodes. We formalized the different entities and concepts which are included in the definition of our model. In particular, we formalized the characteristics of jobs that are submitted to a system, the characteristics of high performance computing hardware resources, and the capabilities and functions which job scheduling policies and resource selection policies provide the whole architecture. We also formalized reservation tables and described how they are used to decide where jobs are allocated and how resources are used.

We stated how results obtained with these new models can differ qualitatively and quantitatively from results obtained using conventional models. Furthermore, we discussed how a consideration of resource selection policies used in scheduling systems can provide different evaluations of job scheduling strategies.

In this chapter we describe two new resource selection policies that are designed to minimize the saturation of shared resources. The first one, the *Find Less Consume Allocation* (henceforth referred to as *LessConsume*) attempts to minimize the job runtime penalty that an allocated job will experience. Based on the utilization status of shared resources in the current scheduling outcome and the job resource requirements, the LessConsume policy allocates each job process to the free allocations where the job is expected to experience an important reduction in the runtime penalties. Note that this RSP puts no restrictions on how long the job start time can be delayed, which means that although the job may have the lowest possible penalty, it could start substantially later than if it had been allocated with a First Fit RSP.

The second once, the *Find Less Consume Threshold Distribution* (henceforth referred to as *LessConsumeThreshold*) , finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. This resource selection policy has been designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job. Thus, the scheduler can ask the RSP to find an allocation for a given job in an iterative process until the most appropriate allocation is found. In the final part of this chapter we will describe the *Resource Usage Aware Backfilling* job scheduling policy. This backfilling-based scheduling policy determines the algorithms for deciding which job has to be moved to the reservation table, and how the jobs have to be backfilled, based on the *LessConsumeThreshold* resource selection policy.

In the previous chapter we described how the resource selection policies First Fit and First Continuous Fit behave. As discussed in the previous chapter, these two selection policies do not take into account job resource requirements (only the requested processors), or how the resources are used. In most of the cases, in the HPC centers, the Local Resource Managers consider two different approaches. The first one consist on using any of the two previous resource selection policies for allocating the jobs. The second one, more frequently used, consists on applying a set of basic filters specified in the user submission script to the available resources when the job allocation is computed. For instance, using the filters the user can require a whole node for executing the application. Furthermore, some of them allow to assure a given amount of resources to the jobs. For instance, using *Constraints* in SLURM (73) or *Consumable resources* in LoadLeveler (68), MOAB (84) and MAUI (83), the user can specify the amount *GB/Sec* of memory bandwidth that the job requires. In the presented work, this approach can be comparable to use the *LessConsumeThreshold* resource selection policy described later with a threshold of *1* (no job penalty is accepted).

## 7.2 The LessConsume Allocation Resource Selection Policy

In this section, we describe the Less Consume resource selection policy whose objective is to provide the allocation that will result in the lowest runtime penalty for jobs due to resource sharing saturation. Using as a basis the First Fit resource selection policy, the main idea is to minimize the penalty associated to the allocation of all job processes evaluating all the different buckets that are included in the space search that is created in the first stage of the algorithm (see First Fit algorithm description of the previous chapter).

The core algorithm of this selection policy is similar to the one described in the First Fit RSP, since the initialization of the algorithm uses the FF selection policy, steps *1*, *2*, *3* and *4* (see 6.2.2), until it reaches a valid allocation. However, in contrast to the previous algorithm, once the base allocation is found (FF) the algorithm computes the penalties associated with the different processes that would be allocated in the reservation. Thereafter it attempts to improve the allocation by replacing the selected buckets (used for create this initial allocation) that would have higher penalties with buckets that are selected in step *1*, but that have not been evaluated. The LessConsume algorithm will iterate until the rest of the buckets have been evaluated or the penalty factor associated to the job is 1 (no penalty). [1]

The LessConsume policy, given a required start time $t_{req}$ and given the job $\alpha$ with a requested runtime of $\alpha_{\{RunTime,rt\}}$ and number of requested processors $\alpha_{\{CPUS,p\}}$, finds in the reservation table the $\alpha_{\{CPUS,p\}}$ processor allocation that tries to minimize the job runtime penalties due to resource sharing saturation closest to the $t_{req}$. To do this, the selection policy follows these steps:

1. For each processor $p_i$ in the reservation table, the resource selection policy selects the interval of time $[t_{x_i}, t_{y_i}]$ that is closest to the $t_{req}$ that satisfies it: no process is allocated to the processor during the interval and its length is equal or greater to $\alpha_{\{RunTime,rt\}}$ . All the buckets [2] associated to the selected intervals of time are added to the set *Buckets* where they are strictly ordered by the interval start time.

[1] The penalty factor is computed:
$PenaltyFactor_{\alpha} = \frac{\alpha_{\{RunTime,rt\}} + \alpha_{\{PenalizedRunTime,prt\}}}{\alpha_{\{RunTime,rt\}}}$

[2] Remember that, we define this interval of time as a *bucket*. Thus the $b_{(i,t_{i_0},t_{i_1})}$ bucket is defined as the interval of time $[t_x, t_y]$ associated to the processor $p_i$

2. Given all the different buckets $\left\{b_{(1,t_{1_0},t_{1_1})},..,b_{(N,t_{N_0},t_{N_1})}\right\}$ associated with the reservation table that are included in the set *Buckets*, the resource selection policy will select the first $\alpha_{\{CPUS,p\}}$ buckets that satisfy the condition that their interval of time shares at least the runtime required by the job.

3. In the case that in step *2* the number of buckets that satisfied the required conditions was lower than the required processors, this implies that there were not enough buckets which shared the required amount of time. In these situations, the first bucket $b_{(i,t_{i_0},t_{i_1})}$ with start time greater than to $t_{req}$ is selected, the $t_{req}$ is updated as $t_{req} = t_{i_0}$ and the steps *1*, *2* and *3* are iterated again.
   At this point, from the buckets obtained in the first step we will have three different subsets:

   - The buckets $\Pi_{disc} = \left\{b_{(k,t_{k_0},t_{k_1})},..,b_{(l,t_{l_0},t_{l_1})}\right\}$ that have already been selected since they cannot form part of a valid allocation for the specified requirements.
   - The buckets $\Pi_{sel} = \left\{b_{(m,t_{m_0},t_{m_1})},..,b_{(n,t_{n_0},t_{n_1})}\right\}$ that have been selected for the base allocation and that conform to a valid allocation which satisfy the requirements for the job. In the case that the penalties of this allocation cannot be improved by a valid allocation, this set of buckets will be used.
   - The buckets $\Pi_{toCh} = \left\{b_{(o,t_{o_0},t_{o_1})},..,b_{(p,t_{p_0},t_{p_1})}\right\}$ that have not already been evaluated. These buckets will be used to try to improve the base allocation. Thus, the LessConsume policy will try to find out if any of these buckets could replace any of the already selected buckets reducing the penalty factor of the job.

4. For each of the buckets in the set of the selected buckets $\Pi_{sel}$, the algorithm checks the penalty that a process of the given job would achieve if it were allocated to the given bucket [3]. Each of the selected buckets has an associated penalty factor. If all the buckets have an associated penalty of *1* the allocation definition based on these buckets is defined and returned. Otherwise, the *last valid allocation* is initialized based on this set of buckets and the selection process goes ahead.

5. For each bucket $b_{(r,t_{r_0},t_{r_1})}$ in the set of buckets $\Pi_{toCh}$:

[3]For more details about how the penalties are computed see the previous chapter. In this case, the penalty of the process is computed by checking the penalty that a job with the same characteristics of the job $\alpha$ but with $\alpha_{\{CPUS,1\}}$ would be allocated using the given bucket

(a) If the number of buckets in the set $\Pi_{sel}$ is lower than the number of requested processors, the bucket is added to the set and the next iteration continues.

(b) Similar to the previous step, the algorithm evaluates the penalty $penalty_r$ that a process of the job would have if this bucket were used to allocate it.

(c) The bucket $bMax_{(p,t_{p_0},t_{p_1})}$ of the set $\Pi_{sel}$ with the highest penalty $penalty_p$ is selected.

(d) In case that the penalty that $penalty_r$ is lower than the penalty $penalty_p$, on one hand the bucket $b_{(r,t_{r_0},t_{r_1})}$ is added to the set $\Pi_{sel}$ and removed from the set $\Pi_{toCh}$. On the other hand, the bucket $bMax_{(p,t_{p_0},t_{p_1})}$ is removed from the $\Pi_{sel}$ and added to the set $\Pi_{disc}$. Note that at this point the inserted bucket may not share the interval required to allocate the job to the rest of the buckets of the set $\Pi_{sel}$.

   i. The buckets of the set $\Pi_{sel}$ that do not share the required time are removed from this set and added to the $\Pi_{disc}$.

   ii. If the number of buckets of the $\Pi_{sel}$ it is the number of requested processors *the last valid allocation* is built based on this set. If the current penalty of all the buckets is 1, the current set of buckets is returned as the allocation. Otherwise the algorithm goes ahead with the next iteration to evaluate the next bucket of the set $\Pi_{toCh}$.

6. The *last valid allocation* is returned.

As an example, suppose that the current scheduling outcome is the one presented in the figure 7.1 and that an allocation has to be found for a job with requested processors $\alpha_{\{CPUS,3\}}$ and runtime $\alpha_{\{RunTime,20secs\}}$.

1. Firstly, in the step *1* the LessConsume algorithm would define the set $\Pi_{toCh} = \{b1-3, b7, b17, b4-6, b11-12, b14-16, b13, b18\}$.

2. In the step *2* the three buckets *1*, *2* and *3* would be selected. Note that all of them satisfy the condition that they all share an interval of time greater than the required runtime.

3. In the step *3* a valid allocation is created with the buckets selected in the previous step. Thus, the set of buckets $\Pi_{sel}$ is composed of $\Pi_{sel} = \{b1, b2, b3\}$. Note that these buckets are removed from the set $\Pi_{toCh}$.

4. In the step *4* the penalty associated with each of the buckets is computed. In the example, due to the resource saturation, the three buckets have an associated

penalty of $penalty_1$,$penalty_2$ and $penalty_3$ greater than one (marked with a gray area in the figure 7.2). With these buckets the basic allocation is created because they share the required amount of time.

5. In step *5* the algorithm has to evaluate whether the current allocation can be improved by replacing any of the buckets of $\{b1, b2, b3\}$ by any of the buckets that have not been evaluated $\Pi_{toCh}$.
   (a) As the current number of buckets in $\Pi_{sel}$ is equal to the requested processors the algorithm continues the iteration.
   (b) The bucket with the maximum penalty is selected *b1* ($> 1$).
   (c) In the first iteration the bucket *b7* is evaluated. The penalty factor that processing the job $\alpha$ would experience if it were allocated using this bucket would be $penalty_7 = 1$.
   (d) As the penalty $penalty_7$ is lower than the $penalty_1$, on one hand the bucket *b1* is removed form the $\Pi_{sel}$ and inserted to $\Pi_{desc}$. On the other hand the bucket *b7* is removed from the $\Pi_{toCh}$ and inserted in $\Pi_{sel}$. The reservation table at this point is shown in figure 7.3. Since not all the penalties of the selected buckets are 1 the algorithm goes ahead with the next bucket.
   (e) As in the first iteration, since the current number of buckets in $\Pi_{sel}$ is equal to the requested processors the algorithm continues the iteration.
   (f) The bucket with the maximum penalty is selected *b2*.
   (g) In the second iteration, bucket *b17* is evaluated. The penalty factor that a process of the job $\alpha$ would experience if it were allocated using this bucket would be $penalty_{17} = 1$.
   (h) As the penalty $penalty_{17}$ is lower than the $penalty_2$, on one hand the bucket *b1* is removed form the $\Pi_{sel}$ and inserted to $\Pi_{desc}$. On the other hand the bucket *b17* is removed from the $\Pi_{toCh}$ and inserted in $\Pi_{sel}$. The reservation table at this point is shown in figure 7.4. Since all the penalties of the selected buckets is 1, the algorithm returns the allocation based on the currently selected buckets, because they provide the minimum penalty.

In the previous example, the start time for the allocation computed using the LessConsume resource allocation policy is the same as would have been obtained by using a First Fit resource selection policy. Thus, in the previous example, the start time remained equal to the First Fit RSP, and the penalty associated with the job has been

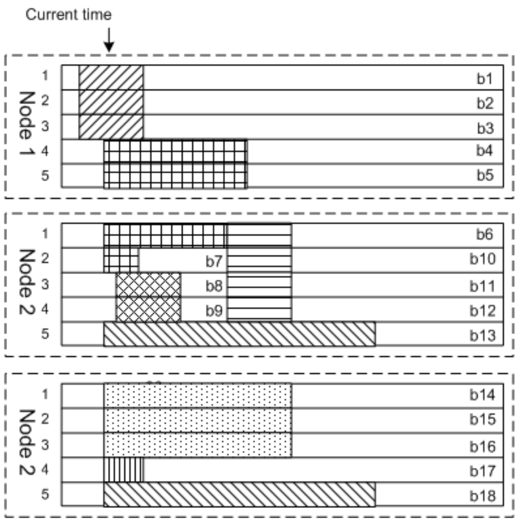


Figure 7.1: Less Consume Example

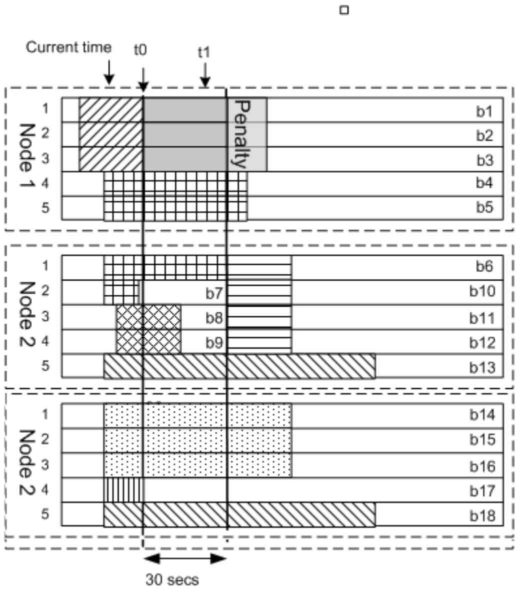


Figure 7.2: Less Consume Example - General Step 1

reduced. However, in some situations, the LessConsume policy may provide allocations with later start times than those using First Fit.

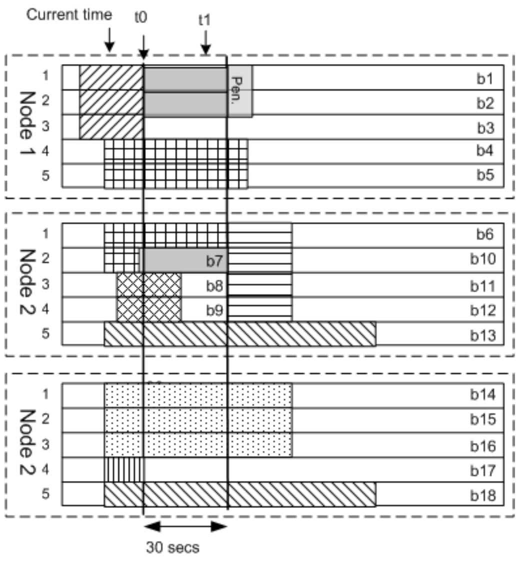


Figure 7.3: Less Consume Example - General Step 2

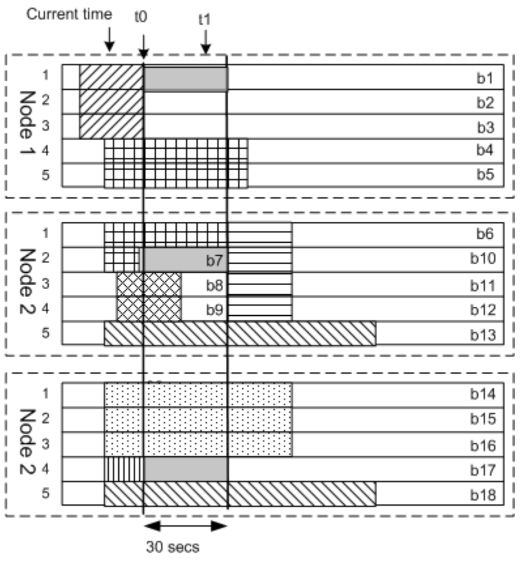


Figure 7.4: Less Consume Example - General Step 3

This last situation is described in the example presented in figures 7.5, 7.6 and 7.7.

Using the previous example, let us suppose that given the scheduling outcome presented in figure 7.5 an allocation has to be found for a job with requested processors $\alpha_{\{CPUS,3\}}$ and runtime $\alpha_{\{RunTime,20secs\}}$. Note that the difference between the two scenarios is that in the initial outcome of the first example, the bucket *b17* starts later in the second example. Thus, in this situation step *g)* the bucket cannot be selected because it does not share the required amount of time with buckets *b1* and *b7*.

The resource selection algorithm for the last example will be the same as in the previous one until step *f)*, and at this point the algorithm will check the rest of the buckets $\Pi_{toCh} = \{b4-6, b10-18\}$. In this example, the next steps are:

1. As the current number of buckets in $\Pi_{sel}$ is equal to the requested processors the algorithm continues the iteration.
2. The bucket with the the maximum penalty is selected *b2*.
3. In the next iteration the buckets *b4* and *b5* will be evaluated (remember that the sets are ordered by the bucket start time). They are discarded due to their penalty is the same as the penalty of *b2*.
4. In the next iteration the bucket *b6* is evaluated. The penalty factor that a process of the job $\alpha$ would experience if it were allocated using this bucket would be $penalty_4 = 1$.
5. As the penalty $penalty_6$ is lower than the $penalty_2$, on one hand the bucket *b2* is removed from the $\Pi_{sel}$ and inserted in $\Pi_{desc}$. On the other hand the bucket *b6* is removed from the $\Pi_{toCh}$ and inserted in $\Pi_{sel}$. However, in this example, as the bucket *b7*, currently in the set $\Pi_{sel}$ does not share the required interval of time with the current evaluated bucket (*b4*), it is removed from this set and inserted in $\Pi_{desc}$.
6. In the next steps the rest of the buckets will be evaluated. In this scenario, none of the jobs of the process $\alpha$ can be allocated to the same node with a penalty one. Therefore, following the presented algorithm, the buckets that will be finally selected are (see figure 7.7) *b3*, *b6* and *b14*.

## 7.3 The Less Consume Threshold Resource Selection Policy

As we have shown in the examples, in some situations this policy not only minimizes the penalized factor of the allocated jobs, but it also provides the same start times as

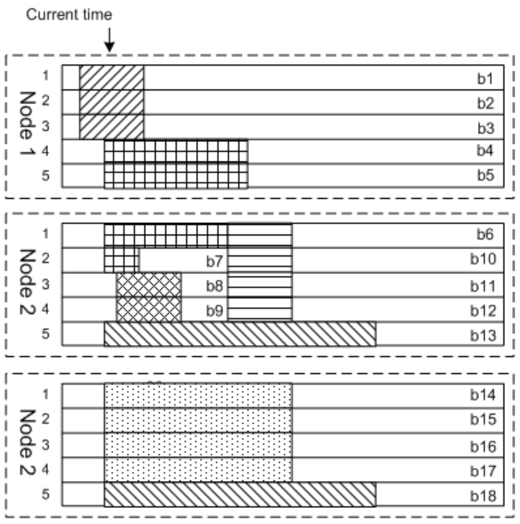


Figure 7.5: Less Consume Example 2

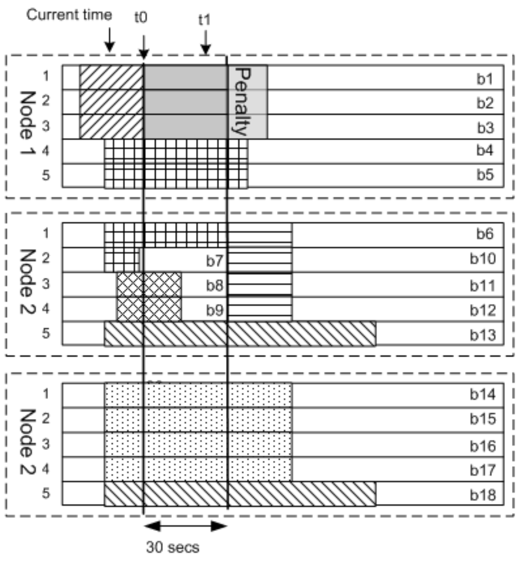


Figure 7.6: Less Consume Example 2 - General Step 1

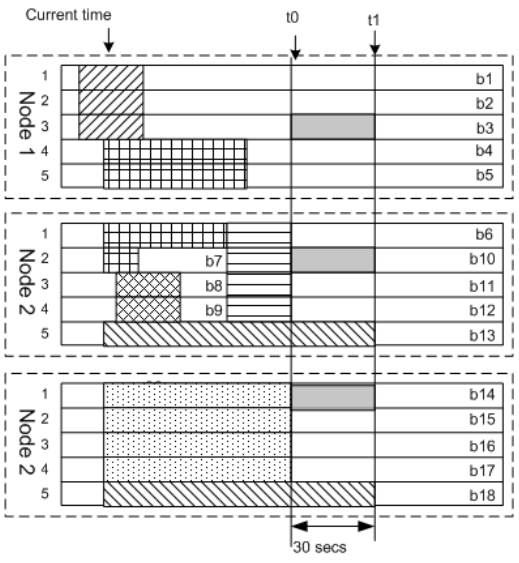


Figure 7.7: Less Consume Example 2 - General Step 3

the first fit allocation policy, which in practice provides the earliest possible allocation start time. However, in many situations (see example 2 above) the allocation policy of the lower penalty factor provides a start time that is substantially later than that achieved by a FirstFit allocation. To avoid circumstances where the minimization of the penalty factor results in delays in the start time of scheduled jobs, we have designed the LessConsumeThreshold RSP. This is a parametrized selection policy which determines the maximum allowed penalty factor allocated to any given job.

This resource selection policy has been mainly designed to be deployed in two different scenarios. In the first case, the administrator of the local scenario specifies in the configuration files the penalty factor of a given allocated job. This factor could be empirically determined by an analytical study of the performance of the system. In the second, more plausible, scenario, the local scheduling policy is aware of how this parametrized RSP behaves and how it can be used by different factors. In this second case the scheduling policy can take advantage of this parameter to decide whether a job should start in the current time or whether it could achieve performance benefits by delaying its start time. In the following subsection we describe a backfilling based resource selection policy that uses the LessConsumeThreshold RSP to decide which jobs should be backfilled and how to allocate the jobs.

The main differences between the two policies is that in the steps *4)* and *5.d.2)* if the number of buckets of the set $\Pi_{sel}$ is the required processors for the job, and they have an associated penalty factor lower or equal to the specified *Threshold*, then the allocation will be defined and returned based on the current set. Note, that in the LessConsume policy the algorithm would iterate evaluating the rest of the free buckets. On the other hand, note that policy enforces that the job allocation penalty must be lower than the provided threshold. Thus if in the step *6)* of the algorithm an allocation is found but has a higher penalty the *treq* will be updated as in the step *4)* of the First Fit and the process would be iterated again. Note that the LessConsume would be stopped at this point due to it has a valid allocation with the lowest penalty that could achieve by optimizing the First First allocation.

## 7.4 The Resource Usage Aware Backfilling Scheduling Policy

The LessConsume policy can be used by any local scheduling policy since it demands schedulers to provide the LRM with the minimum information. In contrast to this first selection policy, the LessConsumeThreshold allows the scheduler or to the administrator to specify the maximum desired penalty factor that the scheduler accepts for a given job. Thus, it is able to carry out the scheduling decisions taking into account the resource sharing saturation and it is able to verify how the job response time is affected by different allocations of the job. Note that the response time of a job can be improved in two different ways:

- Reducing the final runtime of the job by minimizing the penalty factor associated to the job.
- Reducing the wait time of the job by minimizing the start time of the job.

The Resource Usage Aware Backfilling Scheduling (RUA-Backfilling) policy takes into account both considerations when inspecting the wait queue for backfilling the jobs or finding the allocations for the reservations. In brief, this backfilling variant is based on the Shortest-Backfilled First backfilling variant. Rather than backfilling the first job that can moved to the run queue, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalty factors. However, it also takes into consideration the expected response time of the jobs that it evaluates during the backfilling process.

The different parameters of the RUA-Backfilling are:

1. The number of ***reservation*** (number of the jobs in the queue whose estimated start time can not be delayed ) is *1*.

2. The different ***Thresholds*** that will be used to calculate the appropriate allocation for the job that is moved from the reservation. In the evaluation the thresholds used by the policy are $RUA_{thresh} = \{1; 1.15; 1.25; 1.5\}$.

3. The jobs are moved from the wait queue to the reservation using the First Come First Serve priority. This priority assures that the jobs submitted to the system will not suffer starvation.

4. The backfilling queue is ordered using a dynamic criteria that is computed each time that the backfilling processes is required. It is described below.

When a job $\alpha$ has to be moved to the reservation the following algorithm is applied:

1. For each *Threshold* in the $RUA_{thresh}$ specified in the configuration of the policy (in the presented evaluation $\{1; 1.15; 1.25; 1.5\}$):
   (a) The allocation based on the *LessConsumeThreshold* with a parameter of $PenaltyFactor = Threshold$ is requested from the Local Resource Manager.
   (b) The slowdown for the job is computed based on the start/wait times, the penalized runtime of the job in the returned allocation, the current wait time of the job and its requested runtime.

2. The allocation with less slowdown is selected to allocate the job. The local scheduler contacts the local resource manager to allocate the job in the given allocation.

Given the jobs that are queued in the backfilling queue, the backfilling algorithm behaves as follows:

1. In the first step, for each job $\alpha$ in the backfilling queue its allocation is computed based on the algorithm introduced in the previous paragraph. If the start time for the returned allocation is the current time the job is added to the backfilling queue where the allocations are ordered by the penalized factor associated to the allocation and secondly by its length. Note that each job has only one assigned allocation.

2. In the second step, the backfilling queue has all the jobs that could be backfilled in the current time stamp ordered in terms of the associated penalty. The queue is evaluated and the first job that can be backfilled is allocated to the reservation table using allocation computed in the previous step. Note that the allocation will be exactly the same as the one computed in the first step.

(a) If no job can be backfilled the process of backfilling is terminated.

(b) Otherwise, steps 1 and 2 will be iterated again.

## 7.5 Experiments

We evaluate the effect of considering memory bandwidth usage when simulating the Shortest Job Backfilled First policy under several workloads and the LessConsume resource selection policies described in this chapter. We also evaluate the impact of the RUA-Backfilling. For the experiments presented in this section we used the same workloads that are described in the previous chapter. In brief, two different workloads from the Feitelson workload archive (29) were used: the Cornell Theory Center workload and the SDSC Blue Horizon. For each of them we generated different scenarios: with high (High), medium (Med), and low (Low) percentage jobs with high memory demand.

For the analysis of the LessConsume resource selection policies and the RUA-Backfilling job scheduling policy the following configurations were evaluated:

1. The Shortest Job Backfilled First scheduling policy.
2. The two Resources Selection Policies (RSP) were evaluated: **LessConsume**; **LessConsumeThreshold** with four different factors (*1*, *1,15*, *1,25* and *1,5*).
3. Using the job runtime model with resource sharing modelization introduced in the previous chapter.

On the other hand, the RUA-Backfilling policy was evaluated with three different memory pressure workloads for each of the CTC and SDSC centers.

## 7.6 Evaluation

### 7.6.1 The LessConsume and LessConsumeThreshold

Tables 7.1 and 7.2 present the $95_{th}$ percentile and average of the bounded slowdown for the CTC and SDSC centers for each of the three workloads for the FirstFit, LessConsume and LessConsumeThreshold resource selection policy. The last one was evaluated with three different factors: *1*, *1,15*, *1,25* and *1,5*. In both centers the LessConsume policy performed better than the LessConsumeThreshold with a factor of *1*. One could expected that the LessConsume should be equivalent to use the LessConsumeThreshold with a

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 4,2 | 5,94 | 7,92 | 6,12 | 5,32 | 5,23 |
| | Med | 2,8 | 3,55 | 4,22 | 3,82 | 3,65 | 3,52 |
| | Low | 2,2 | 3,12 | 3,62 | 3,82 | 3,45 | 3,52 |
| SDSC | High | 99,3 | 110,21 | 128,08 | 115,28 | 109,51 | 106,23 |
| | Med | 55,4 | 68,06 | 74,32 | 72,83 | 71,37 | 68,52 |
| | Low | 37,8 | 45,37 | 57,27 | 52,86 | 42,28 | 42,28 |

Table 7.1: Bounded-Slowdown - $95_{th}$ Percentile

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 8,2 | 10,44 | 18,02 | 12,38 | 11,32 | 13,54 |
| | Med | 5,3 | 6,65 | 7,52 | 8,05 | 6,85 | 7,75 |
| | Low | 3,2 | 5,62 | 5,92 | 5,82 | 8,84 | 8,56 |
| SDSC | High | 22,56 | 24,41 | 27,37 | 25,54 | 24,26 | 23,53 |
| | Med | 11,32 | 12,51 | 14,76 | 14,46 | 14,17 | 13,6 |
| | Low | 7,54 | 7,8 | 9,08 | 9,5 | 8,47 | 8,27 |

Table 7.2: Bounded-Slowdown - Average

threshold of *1*. However, note that this affirmation would be incorrect. This is caused due to the LessConsume policy at the step *6)* of the presented algorithm will always stop. The goal of this policy is to optimize the First Fit allocation but without carry out a deeper search of other possibilities. However, the LessConsumeThreshold may look further in the future in the case that the penalty is higher than the provided threshold. Thereby, this last one is expected to provide higher wait time values. On the other hand, as we had expected, the bounded slowdown decreases while increasing the factor of the LessConsumeThreshold policy. In general, the ratio of increment of using a factor of *1* and a factor of *1,5* is around a 20% in all the centers and workloads.

The performance of these two resource policies, compared to the performance of the First Fit policy discussed in the previous chapter, shows that LessConsume policies give an small increment in the bounded slowdown. For instance, in the CTC high memory pressure workload the $95_{th}$ percentile of the bounded slowdown has increased from 4,2 in the First Fit to 5,94 in the LessConsume policy, or to 7,92 and 5,23 in the LessConsumeThreshold with thresholds of *1* and *1,5* respectively.

Tables 7.8 and 7.4 show the $95_{th}$ and average of the wait time for the CTC and

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 10286 | 12588 | 17945 | 15612 | 14555 | 10188 |
| | Med | 8962 | 9565 | 13391 | 13123 | 9186 | 9094 |
| | Low | 4898 | 5034 | 6544 | 7198 | 5235 | 5759 |
| SDSC | High | 55667 | 63293 | 70964 | 69632 | 59978 | 41779 |
| | Med | 44346 | 45164 | 58713 | 59300 | 47440 | 45616 |
| | Low | 32730 | 35092 | 38265 | 37499 | 33374 | 33785 |

Table 7.3: Wait Time - $95_{th}$ Percentile

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 20082 | 24576 | 35035 | 30480 | 28416 | 19890 |
| | Med | 13443 | 14347 | 20086 | 19684 | 13779 | 13641 |
| | Low | 7124 | 7322 | 9518 | 10469 | 7614 | 8376 |
| SDSC | High | 12647 | 14379 | 16122 | 15819 | 13626 | 9491 |
| | Med | 9061 | 9228 | 11996 | 12116 | 9693 | 9320 |
| | Low | 3931 | 4214 | 4595 | 4503 | 4008 | 4057 |

Table 7.4: Wait Time - average

SDSC centers for each of the three workloads for the FirstFit, LessConsume and LessConsumeThreshold resource selection policy. These performance variable show similar pattern to the bounded slowdown. The LessConsume policy shows a better performance result that using the LessConsumeThreshold with a factor of *1*.

The $95_{th}$ percenage of penalized runtime is presented in the table 7.5 and the average

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 8,8 | 8,01 | 7,69 | 7,87 | 7,91 | 8,1 |
| | Med | 4,8 | 3,81 | 3,01 | 3,52 | 4,06 | 3,90 |
| | Low | 0,92 | 0,78 | 0,51 | 0,72 | 0,62 | 0,80 |
| SDSC | High | 11,8 | 11,33 | 8,31 | 10,37 | 11,58 | 11,64 |
| | Med | 6,7 | 6,01 | 4,70 | 4,85 | 5,64 | 5,96 |
| | Low | 1,4 | 1,03 | 0,75 | 0,81 | 0,94 | 1,19 |

Table 7.5: Percentage of Penalized Runtime - $95_{th}$ Percentile

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 7,8 | 6,81 | 6,98 | 7,17 | 7,39 | 7,34 |
| | Med | 3,8 | 2,54 | 2,93 | 3,02 | 3,1 | 3,20 |
| | Low | 0,72 | 0,7 | 0,21 | 0,42 | 0,33 | 0,64 |
| SDSC | High | 15,1 | 10,32 | 7,41 | 11,73 | 10,85 | 12,32 |
| | Med | 10,2 | 7,2 | 4,70 | 4,85 | 5,64 | 5,96 |
| | Low | 5,2 | 4,53 | 2,56 | 3,11 | 4,58 | 4,23 |

Table 7.6: Percentage of Penalized Runtime - average

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 428 | 120 | 57 | 70 | 87 | 97 |
| | Med | 247 | 101 | 76 | 77 | 102 | 99 |
| | Low | 64 | 45 | 36 | 38 | 58 | 52 |
| SDSC | High | 475 | 105 | 87 | 130 | 127 | 130 |
| | Med | 255 | 89 | 76 | 79 | 103 | 145 |
| | Low | 51 | 34 | 22 | 27 | 33 | 41 |

Table 7.7: Number of Killed Jobs $95_{th}$ Percentile

is shown in the table table 7.6. The penalized runtime clearly increases by incrementing the threshold. For instance, the $95_{th}$ Percentile of the percentage increases from 8,31 in the SDSC and the high memory pressure workload with a factor of *1* until 11,64 with a factor of *1,5*. The LessConsume, different from to the two previously described variables, shows similar values to the LessConsumeThreshold with a factor of *1,5*. This percentage of penalized runtime was reduced with respect to the First Fit when using all the different factors in both centers.

The number of killed jobs is the performance variable that showed most improvement in all the memory pressure workloads. The number of killed jobs is qualitatively reduced with the LessConsumeThreshold with a factor of *1*: for example with the high memory pressure workload and the CTC center, the number of killed jobs was reduced from 428 with the First Fit to 70. The other threshold factors also showed clear improvements; the number was halved. As to the LessConsume policy, the number of killed jobs was reduced by a factor of 4 compared to the First Fit and the high and medium memory pressure workloads of both centers.

The performance variables above show the performance that jobs experienced with

| Center | MEM | FF | LC | LCT=1 | LCT=1,15 | LCT=1,25 | LCT=1,5 |
|---|---|---|---|---|---|---|---|
| CTC | High | 412 | 400 | 372 | 382 | 367 | 385 |
| | Med | 411 | 412 | 343 | 375 | 389 | 399 |
| | Low | 412 | 412 | 345 | 321 | 373 | 399 |
| SDSC | High | 111 | 103 | 91 | 101 | 104 | 101 |
| | Med | 110 | 105 | 87 | 105 | 102 | 110 |
| | Low | 110 | 108 | 89 | 107 | 106 | 107 |

Table 7.8: Avg. CPUS Used/Hour $95_{th}$ Percentile

the LessConsume and LessConsumeThreshold resource selection policies. The table 7.8 presents the $95_{th}$ Percentile of the average CPUS Used/Hour during the simulation for the LessConsume and LessConsumeThreshold for each of the memory pressure workloads and thresholds. Clearly, the LessConsume policy showed better use of the resources of the system. The LessConsumeThreshold shows higher utilization when the penalty factor is increased.

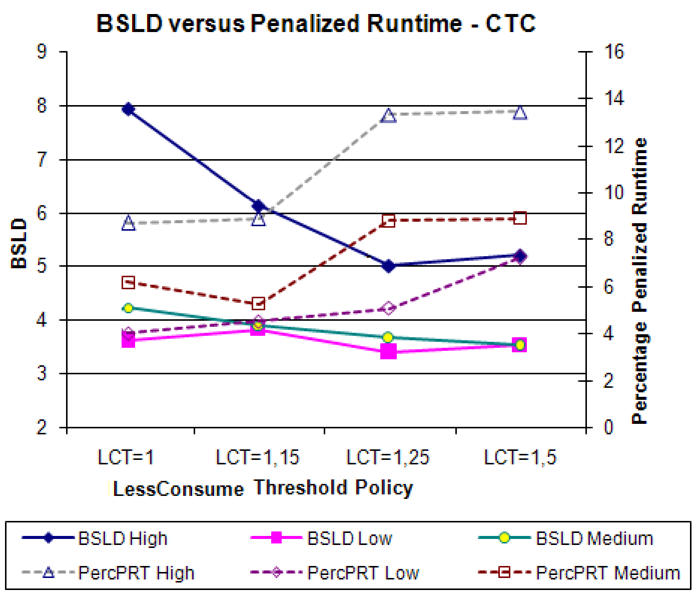


Figure 7.8: BSLD versus Percentage of Penalized Runtime - CTC Center

The LessConsume policy shows how the percentage of penalized runtime and number of killed jobs can be reduced in comparison to the First Fit and First Continuous Fit, by using this policy with EASY backfilling. In traditional scheduling architectures this RSP can be used rather than traditional policies, without any modifications in local scheduling policies. Furthermore, the LessConsume threshold shows how, with different thresholds, performance results can also be improved. Relaxing the penalty factor results in better performance of the system, and in an increase in the number of killed jobs and the percentage of penalized runtime. The LessConsume policy shows similar performance results as the LessConsumeThreshold with factors of *1,25* and *1,5*.

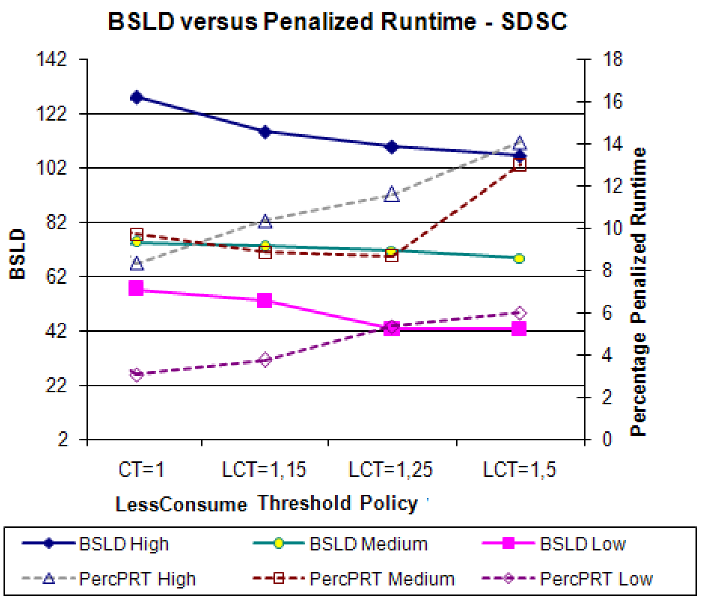


Figure 7.9: BSLD versus Percentage of Penalized Runtime - SDSC Center

Figures 7.8, 7.9, 7.10 and 7.11 present the performance of the LessConsume policies (using bounded slowdown) against the percentage of penalized runtime of the jobs and the number of killed jobs. The goal of these figures is to show the chance that the LessConsumeThreshold and LessConsume policies have to improve the performance of the system while achieving an acceptable level of performance. As can be observed in figures 7.10 and 7.8 a good balance is achieved in the CTC center using the threshold of *1,15* where both the number of Killed Jobs and the percentage of penalized runtime converge are in acceptable values. In the case of the SDSC center, this point of convergence is not as evident as the CTC center. Considering the tendency of the bounded slowdown, it seems that the LessConsumeThreshold with a factor of *1* is an appropriate configuration for this center, due to the fact that the penalized runtime and the number of killed jobs presents the lowest values, and the bounded slowdown shows values that are very close to the factors of *1,15* and *1,25*. However, the configuration of the LessConsumeThreshold with a factor of *1,15* also shows acceptable values.

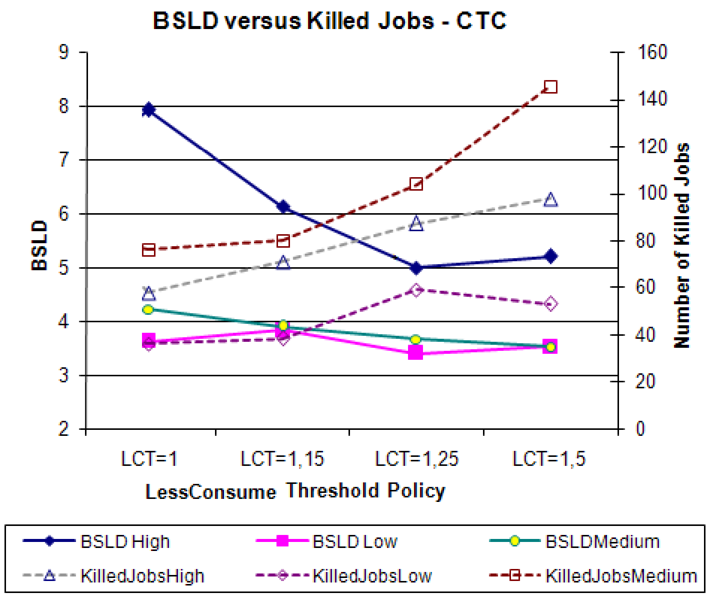


Figure 7.10: BSLD versus Killed Jobs - CTC Center

### 7.6.2 The RUA-Backfilling

The figure 7.12 presents the performance that the RUA-Backfilling scheduling policy has achieved with respect the Shortest-Job-Backfilled First Backfilling (SJBF-Backfilling) with the First Fit and LessConsume Resource Selection Policies. The results shows also how each of the policies behaved with the three different memory pressure workloads for the SDSC and CTC workloads. The figure shows the $95_{th}$ Percentile of the BSLD, the Wait time and the Percentage of Penalized Runtime that the jobs have experimented and the number of killed jobs that three scheduling strategies have achieved in the

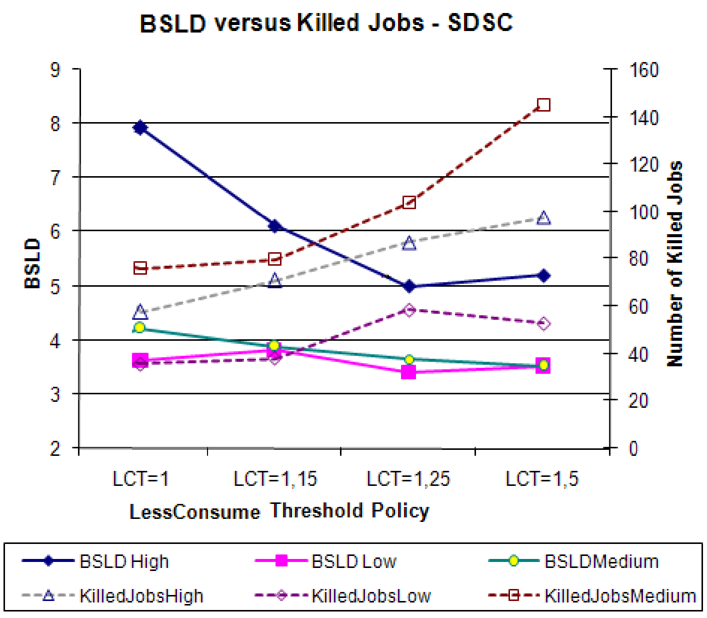


Figure 7.11: BSLD versus Killed Jobs - SDSC Center

simulations.

The bounded slowdown shows how in both workloads the RUA-Backfilling achieves slightly worst performance that the SJBF-Backfilling with the FF selection policy. In both cases the difference between the BSLD is less than a 5%. For instance, the $95_{th}$ Percentile of the BSLD with the SJBF-Backfilling in the SDSC workload with high memory pressure is 100 and with the RUA-Backfilling is around 110. Note, that we could expect that this last one should achieve slower BSLD that the once obtained by the SJBF-Backfilling with FF due to it takes into account the resource usage. However, in the RUA-Backfilling the number of jobs that are used for compute the BSLD (number of finished jobs) is substantially bigger that the once used in the other (400 less in the SJBF). Respect the SJBF-Backfilling with the LessConsume resource selection policy, the RUA-Backfilling shows in both workload better bounded slowdowns.

The wait time shows similar patterns that the Bounded Slowdown. However, the CTC workload shows higger differences between the SJBF-Backfilling and the other two strategies. For example, while the $95_{th}$ Percentile of wait time for the RUA-Backfilling and the SJBF-Backfilling with FF remains around 4000 and 5000 seconds in the high pressure scenario, the SJBF-Backfilling with LessConsume presents $95_{th}$ Percentile of the wait time around 7000. This, may indicate that the RUA Backfilling is more stable than using the LessConsume with a non resource usage aware scheduling strategy.

Finally, the number of killed jobs and the $95_{th}$ Percentile of percentage of penalized runtime show a qualitative improvement respect the SJBF-Backfilling with First Fit. For example, the RUA-Backfilling shows an reduction of a 500% in the number of killed jobs in the high memory pressure scenario of the SDSC workload and a reduction of 300% in the CTC scenario also with the high memory pressure scenario. Although the percentage of penalized run time shows an improvement in both center using the RUA-Backfilling, a higher improvement is shown in the SDSC center. For example, in this last case the percentage of penalized runtime is reduced a 50% in the workload with a medium memory pressure.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the Local Resource Manager and the Scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with LessConsume scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

## 7.7 Conclusions

In this chapter we have shown how the performance of the system can be improved by considering resource sharing usage and job resource requirements in two levels of local scheduling scenarios: two new Resource Selection Policies (RSP) and one scheduling policy.

We have described the *Find Less Consume Distribution* that attempts to minimize the job runtime penalty that an allocated job will experience. Based on the utilization status of the shared resources in current scheduling outcome and job resource requirements, the LessConsume policy allocates each job process to the free allocations in which the job is expected to experience the lowest penalties. We have also described the *Find Less Consume Threshold Distribution* selection policy which finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. This resource selection policy has been designed to provide a

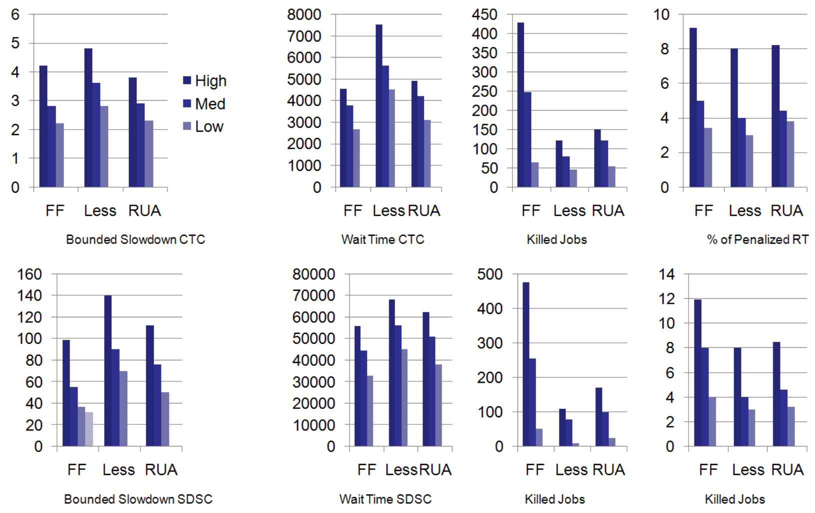


Figure 7.12: RUA Performance Variables

more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job. Thus, this RSP can be used by the scheduler to find an allocation for a given job in an iterative process until the most appropriate allocation is found.

In the final part of this chapter we have described the *Resource Usage Aware Backfilling* job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be moved to the reservation table and how jobs have to be backfilled are based on the *LessConsumeThreshold* resource selection policy.

We have evaluated the impact of using the LessConsume and LessConsumeThreshold (Thresholds *1*, *1.15*, *1.25* and *1.5*) with the Shortest Job Backfilled first. In this evaluation, we have used the workloads described in the previous chapter where we evaluated the impact of memory bandwidth sharing on the performance of the system. Both resource allocation policies show how the performance of the system can be improved by considering where the jobs are finally allocated. The bounded slowdown of both policies show slightly higher values than those achieved by a First Fit resource allocation policy. However, they show a very important improvement in the percentage of penalized runtimes of jobs, and more importantly, in the number of killed jobs, showing a very good balance in the increment of the BSLD. Both have reduced by four or even six times the number of killed jobs in all the evaluated workloads. Note that each of the indicated thresholds depends on the center. In the SDSC the a threshold of *1* or *1.15* shows a good balance of performance (BSDL) and number of killed jobs and percentage of penalized runtime, and in the CTC center the appropriate threshold is *1.5*.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the Local Resource Manager and the Scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with LessConsume scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

Chapter 8

# A Job Self-Scheduling Policy for HPC Infrastructures

Several centralized scheduling solutions have been proposed in the literature for distributed high performance computing architectures, such as centralized schedulers, centralized queues and global controllers. These approaches use a unique scheduling entity responsible for scheduling all the jobs that are submitted by the users.

In this Thesis we propose the usage of self-scheduling techniques using the ISIS-Dispatcher for dispatching the jobs that are submitted to a set of distributed computational hosts that are managed by independent schedulers (such as MOAB or LoadLeveler). It is a non-centralized and job-guided scheduling policy whose main goal is to optimize the job metrics, such as the wait time, slowdown or economic cost. Thus, the scheduling decisions are done independently for each job instead of using a global policy where all the jobs are considered. On top of this, as a part of the proposed solution, we also demonstrate how the usage of job wait time and runtime prediction techniques can substantially improve the performance obtained in the described architecture.

## 8.1 Introduction

The increasing complexity of local systems has led to new more complex scheduling architectures and techniques. The new systems are being built with multiple computational resources with different characteristics and policies. In these new scenarios, the traditional scheduling techniques have evolved to more complex and sophisticated approaches where other issues, such the heterogeneity of the resources (46) or the geographical distribution (26), have been taken into account.

The Multi-site HPC architectures are usually composed of several centers containing many hosts managed by different schedulers (such as LoadLeveler or MOAB). Centralized scheduling strategies have been proposed to manage the jobs that are submitted to these scenarios. Thus, the users submit the jobs to a centralized scheduler. Once the scheduler receives the job submission, it queues the job. It applies a global scheduling policy taking into account all the queued jobs and the resources available in the centers to decide which jobs have to be submitted, where and when. When the resource where the job has to run is selected by the global scheduler, the job is removed from the wait queue and it is submitted to the scheduler that manages the selected resource.

Similar to the philosophy of the AppLeS project (6), in this Thesis we propose using neither a global scheduler, nor global structures to manage the jobs submitted to these scenarios. We propose instead using self-scheduling techniques to dispatch the jobs that are submitted to the set of distributed hosts. In this architecture, the jobs are self-scheduled and there are no centralized scheduling decisions. The dispatcher is aware of the status of the different resources that are available for the job, but it is not aware of the rest of the jobs that other users have submitted and not yet dispatched to the system. Thus, the jobs themselves decide which is the most appropriate resource to execute them. We also propose the use of wait time prediction techniques to select the computational resources.

The target architectures of our work are Mutli-site systems where each computational resource is managed by an independent scheduler (such as MOAB or SLURM). In contrast to the AppLes approach, we propose an interaction between the job dispatcher and the local schedulers. Thus, our Thesis proposes the use of two scheduling layers: at the top, the job is scheduled by the dispatcher (the schedule is based on the information provided by the local schedulers and their capabilities); and, once the resource is selected and the job submitted, the job is scheduled by the local resource scheduler.

For this purpose, we have designed the ISIS-Dispatcher that is a part of the job (see figure 8.1). Once the user submits the job, the dispatcher submits it to the scheduler that controls the resource which is best matched to the job requirements. The ISIS-Dispatcher has been designed to be deployed in large systems, for instance groups of HPC Research Centers or groups of universities. The core of the ISIS-Dispatcher algorithm is based on task selection policies. We also propose four new task selection policies oriented to improve the job wait time (Less-WaitTime and Less-WaitTime-pred) and the job slowdown (Less-Slowdown and Less-Cost-Sld) based on the job wait time and runtime predictions. The main advantage of these new task selection policies is that they take into account the capacity of the available resources while the other policies do not (i.e.: Less-Work-Left, Less-Queued-Jobs etc.).

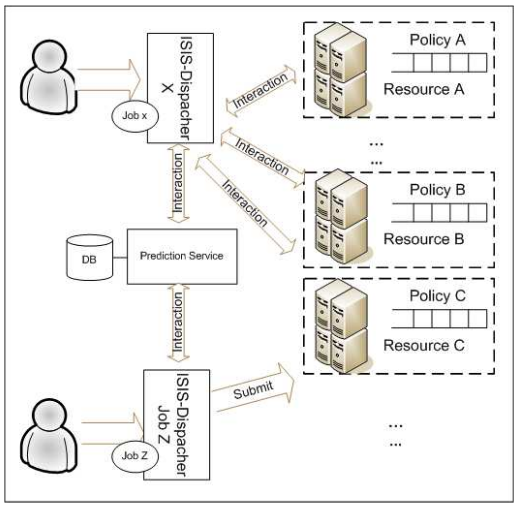


Figure 8.1: The ISIS-Dispacher architecture

In this chapter we present two different evaluations of the proposed architecture. First, we evaluate different task assignment policies proposed in the literature and the Less-WaitTime policy to validate the proposal. The evaluation of the presented architecture shows how the self-scheduling policy can achieve good performance results (in terms of resource usage and job performance). Furthermore, we state how using prediction techniques for the wait time using the job user runtime estimate used in the new Less-WaitTime policy can substantially improve overall performance. The main reason for this improvement is that the use of the new techniques takes into account not only the status of the resource but also its capacity (i.e.: number of processors), while the original techniques only considered the status of the resources (i.e.: the number of queued jobs) and were designed for homogeneous architectures with resources with the same configurations.

In the second we part describe the whole set of the ISIS-Dispatcher task assignment policies (including the Less-Waittime, Less-Waittime-pred, Less-Slowdown and Less-Cost-Sld. We also present their evaluation. This evaluation is done using a new evolutive prediction model based on classification trees and discretization techniques. Furthermore, we introduce the modelling of the economic cost of using different computational resources. The evaluation of the new proposed techniques includes a study of the economic impact of each of the proposed strategies. In this second evaluation we state how the performance of the system can be improved by trying to minimize job slowdown, using job run and wait time predictions. We also state how the Less-Cost-Sld task assignment policy it is able to improve the economic cost related to job execution and the related to the execution of the whole workload.

The rest of the chapter is organized as follows: in the sections Background and Contribution we present the background of the presented work and our main contributions; in the section ISIS-Dispacher we describe the scheduling architecture proposed; in the following section we present the first evaluation of this architecture, including the wait time prediction methodology, the description of Less-Waittime and its evaluation compared with the other proposed policies; next we present the rest of task assignment policies that have been designed for the ISIS-Dispatcher, the runtime prediction model and the economic model used in the architecture and their evaluation; and finally, in the section Summary the summary of the chapter are presented.

## 8.2 Background

### 8.2.1 Multi-site scheduling strategies

The discussed works in the first chapters regarding the local scheduling scenarios have analyzed how local centers behave when jobs are submitted to a specific host managed by one scheduler. In such conditions jobs are executed in the host to which they were submitted. However, in the current HPC centers, they may have many hosts managed by one centralized scheduler, or even more than one host managed by independent schedulers. In these cases, there is the possibility that a job submitted to a Host A could start earlier in Host B of the same center, or even that it could achieve more performance (i.e.: improving the response time) in another Host C. In recent years, scheduling research activities have started to focus on these new scenarios where many computational resources are involved in the architectures.

In the coming large distributed systems, like grids, more global scheduling approaches are being considered. In these scenarios users can access a large number of resources that are potentially heterogeneous, with different characteristics and different access and scheduling policies (see Figure 8.2). Thus, in most cases the users do not have enough information or skills to decide where to submit their jobs. Several models have been proposed in the literature to solve the challenges open in these architectures.

In (103), Yue proposes to apply a global backfilling within a set of independent hosts where each of them is managed by an independent scheduler (Model 1, Figure

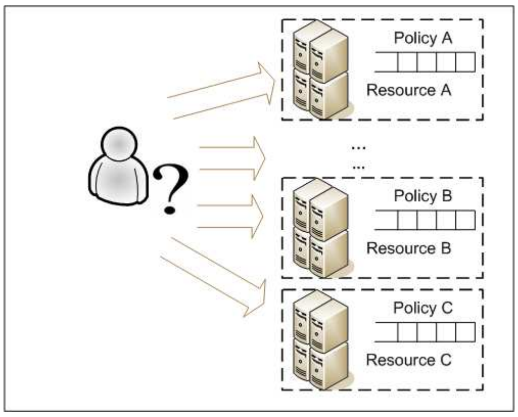
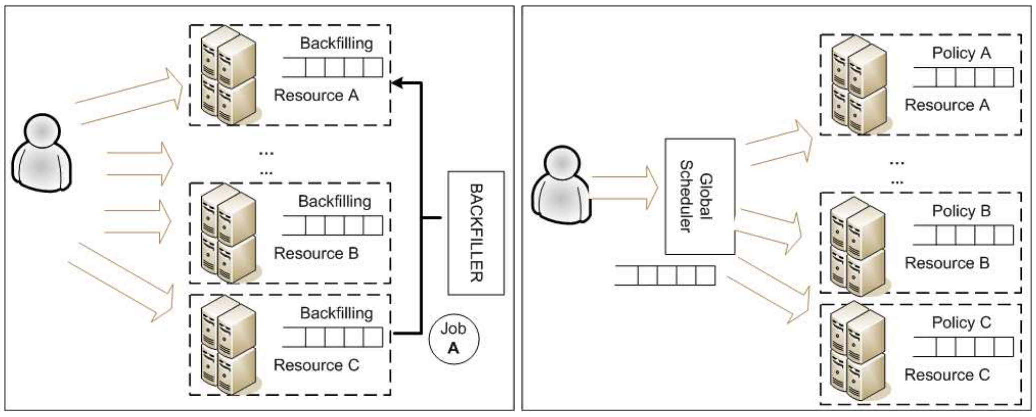


Figure 8.2: Heterogeneous multi-site architectures

8.3a). The core idea of the presented algorithm is that the user submits the jobs to a specific system, managed by an independent scheduler. A global controller tries to find out if the job can be backfilled to another host of the center. In the case that the job can be backfilled in another host before it starts, the controller will migrate the job to the selected one. As the algorithm requires the job runtime estimation provided by the user, this optimization is only valid in very homogeneous architectures. This solution may not scale in systems with a high number of computational hosts. Furthermore, other administrative problems may arise, for instance it is not clear if the global *backfiller* presented could scan the queues of all the host involved in the system due to security reasons or VOs administration policies (41).

Sabin et al. studied in (46) the scheduling of parallel jobs in a heterogeneous multi-site environment (Model 2, Figure 8.3b). They propose carrying out a global scheduling within a set of different sites using a global meta-scheduler where the users submit the jobs. Two different resource selection algorithms are proposed: in the first one the jobs are processed in order of arrival to the meta-scheduler, each of them is assigned to the site with the least instantaneous load; in the second one when the job arrives it is submitted to *K* different sites (each site schedules according to a conservative backfilling policy), once the job is started in one site the rest of the submissions are canceled (this technique is called multiple requests, MR).

In (26) they analyze the impact of geographical distribution of Grid resources on machine utilization and the average response time. A centralized Grid dispatcher that controls all the resource allocations is used (Model 3, Figure 8.4a). The local schedulers are only responsible for starting the jobs after the resource selection is made by the Grid



(a) Global Backfilling - Model 1 (b) Global Scheduler - Model 2

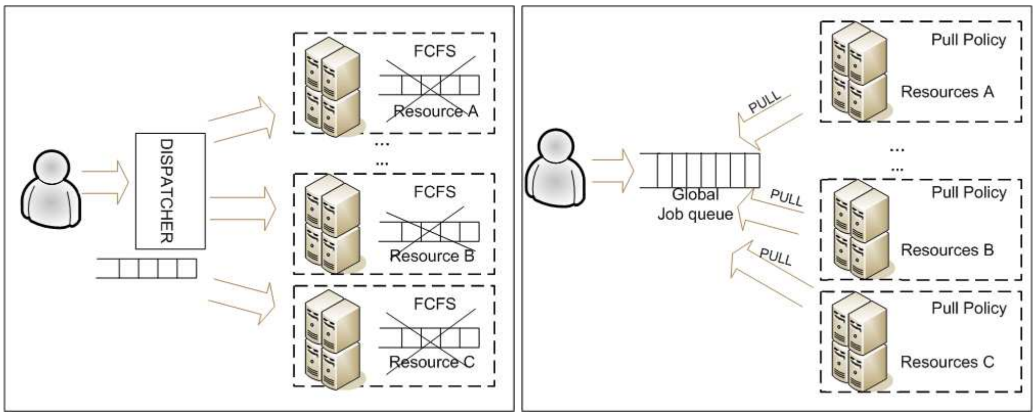
Figure 8.3: Proposed solutions (I)

Scheduler. Thus, all the jobs are queued in the dispatcher while the size of the job wait queues of the local centers is zero. In this model, a unique global reservation table is used for all the Grid and the scheduling strategy used consists of finding the allocation that minimizes the job start time. A similar approach is the one presented by Schroeder et al. in (86), where they evaluate a set of task assignment policies using the same scenario (one central dispatcher).

In (82) Pinchak et al. describe a metaqueue system to manage the jobs with explicit workflow dependencies (Model 3 , Figure 8.4b). In this case, a centralized scheduling system is also presented. However the submission approach is different from the one discussed before. Here the system is composed of a user-level metaqueue that interacts with the local schedulers. In this scenario, instead of the push model, in which jobs are submitted from the metaqueue to the schedulers, placeholding is based on the pull model in which jobs are dynamically bound to the local queues on demand.

In the previously discussed works, using the global policies, the utilization of the available computational resources have been increased. Furthermore, the service received by the users has also been improved. However, in very large domains these approaches may not scale. Therefore, implementing a centralized scheduling algorithms in these architectures is not appropriate. In intervals with high computational demand the global schedulers may not be able to react due to the large number of requests and available centers.

In the AppLess project (6)(5) Berman et al. introduced the concept of application-



(a) Global Dispatcher - Model 3 (b) Global queue using the Pull mechanism - Model 4

Figure 8.4: Proposed solutions (II)

centric scheduling in which everything about the system is evaluated in terms of its impact on the application. Each application developer schedules their application so as to optimize their own performance criteria without regard to the performance goals of other applications which share the system. The goal is to promote the performance of an individual application rather than to optimize the use of system resources or to optimize the performance of a stream of jobs. In this scenario the applications are developed using the AppLess framework and they are scheduled by the Apples agents. These agents do not use the functionalities provide by the resource management systems. Therefore, they rely on systems such as Globus (40), Legion (51), and others to perform that function.

In brief, all the strategies that have been presented in the literature to solve the challenges open in these architectures can be classified in the following models:

- Model 1: There are $K$ independent systems with their own scheduling policies and queuing systems, and one or more external controllers. In this scenario users submit jobs to the specific host with a given policy, and a global scheduling entity tries to optimize the overall performance of the system and the service received by the users. For example, as is exemplified in figure 8.3a the controller may decide to backfill jobs among the different centers (103).

- Model 2: There is a centralized global scheduler that manages the jobs at the global level and at local level schedulers and queuing systems are also installed. In this situation the users submit jobs to the centralized scheduling system that will later submit the job to the selected scheduling system of a given center. Jobs are queued at the two different levels: first at the global scheduler queue and second at the local scheduler queue (see Figure 8.3b). This is the typical brokering approach.

- Model 3: There is a centralized dispatcher that schedules and manages all the jobs but no local schedulers are installed. The local computational nodes only carry out the resource allocation since all the scheduling decisions are taken by the centralized dispatcher. The jobs are queued only at the upper level. (see Figure 8.4a).
- Model 4: There is one centralized global queue where all the jobs are queued. The local computational schedulers pull jobs from the global queue when there are enough available resource for run them. In this way the scheduling decisions are done independently at the local level. In this situation (see Figure 8.4b) the users submit jobs to this centralized queue.

To summarize, all the previous scenarios have two common characteristics:

- The scheduling policies are centralized, the scheduler is aware about all the jobs submitted to the system. Thus, the users submit the jobs to a global scheduler.
- They assume that the local resources are homogeneous and are scheduled according the same policy or even without policy.

Our proposal will be deployed in scenarios with the following conditions:

- The users can submit the jobs directly to the computational resources. Also, they should be able to submit the jobs using the described dispatching algorithm.
- The computational resources can be heterogeneous (with different number of processors, computational power etc.). Also, they can be scheduled by any run-to-completion policy.
- The local resource managers have to provide functionalities for access to information concerning their state (such as number of queued jobs).

### 8.2.2 Task assignment policies

The scheduling policy presented in this Thesis uses task assignment policies to decide where the jobs should be submitted. The subject of job or task assignment policy has been studied in several works and several solutions have been proposed by the research community. Some of the task assignment policies that have been used in the literature are:

- The *Random* policy. The jobs are uniformly distributed among all the available clusters.
- The *Less-SubmittedJobs* policy. The jobs are submitted following a Round Robin distribution.
- The *Less-JobsInQueue* policy. The jobs are submitted to the host with the least queued jobs.
- The *Least-WorkLeft* policy. The jobs are submitted to the host with the least pending work. .
- The *Central-Queue* policy. The jobs are queued in a global queue. The hosts pull the jobs from the global queue when enough resources are available.
- The *SITA-E* policy (proposed in (61)). The jobs are assigned to the host based on their runtime length. Thus, *short* jobs would be submitted to *host 1*, *medium* jobs to *host 2* and so on. This policy uses the runtime estimation of the job. In this case the duration cutoffs are chosen so as to equalize load.
- The *SITA-U-opt* policy (proposed in (86)). It purposely unbalances load among the hosts, and the task assignment is chosen so as to minimize the mean slowdown.
- The *SITA-U-fair* policy (also proposed in (86)). Similar to the *opt*, it bases the assignment to unbalance the host load. However, the goal for this policy is to maximize fairness. In this SITA variant, the objective is not only to minimize the mean slowdown, but also to balance the slowdown for large jobs equal to short jobs.

The evaluations presented in (86)(61) concerning the performance of all these task assignment policies have shown that the SITA policies achieve better results. Schroeacher et al. stated that the Random policy performs acceptably for low loads, but for high loads, however, the average slowdown explodes. Furthermore, the SITA-E showed substantially better performance than Least-Work-Left and Random policies for high loads. However, the Least-Work-Left showed lower slowdown than the SITA-E. SITA-U policies showed better results than the SITE-E. SITA-U-fair improved the average slowdown by a factor of 10 and its variance by a factor of 10 to 100. Harchol presented similar work in (4), where the same situation is studied, however the presented task policy does not know the job duration. Although both SITA-U and SITA-E policies have shown promising performance results, they cannot be used for the Self-Scheduling policy described in this chapter due to the fact that they assume having knowledge of the global workload and they are designed for centralized approaches.

### 8.2.3 Data mining and prediction techniques

Data mining can provide the dispatcher with estimates which allow it to make more intelligent scheduling decisions. In our previous studies (48)(59) data mining was used to correlate past executions of similar jobs using similar resources, or of similar future load using decision trees for the prediction algorithm. In the literature several prediction methodologies have been proposed. However, as will be discussed below, almost all of them have concentrated on predicting job performance variables (such as runtime) for local environments or sites. In (48) we proposed a set of prediction techniques that provide the users with hints about where to submit jobs given a Grid architecture. For example, we estimate how long a given job will wait in a given scheduling system before being executed with specific job requirements (the number of processors, the input and output files etc.).

Papers like those presented by Dinda in (21), propose the use of linear mathematical models in order to predict the runtime of applications submitted by users. Dinda proposes the use of time series (AR, MA ARMA and ARIME) and a windowed mean to carry out host load estimates. These estimates are used to predict job runtime. Yuanyuan presented in (104) new models, also based on time series, to predict runtime for Grid applications. Other works (20) have proposed the use of a state-transition model to characterize the resource use of each program in previous executions.

In (23), Downey characterizes the applications by describing the speedup of the application on a family of curves that are parameterized by a job's average parallelism and its variance. Using this profile, Downey generates the speedups for the job models that will be used to derive the runtime for the applications.

Another commonly used statistical approach is simulation. An example is the Dimemas simulator developed in the Barcelona Supercomputing Center (66). The simulator reconstructs the execution trace file by estimating the time needed to execute each computation burst and communication burst.

Data mining techniques have become very popular in recent years. They are being used in a very wide range of areas, including job performance prediction. Warren Smith et al. in (89) presented a first study of these techniques. Their paper is based on Gibbons' earlier work (47), which consisted of using a static clustering of workloads, and later, of using the mean and median inside these clusters.

The problem of predicting how long a given job will wait in the queue of a system has been explored by other researchers in several papers. (23)(89)(90). A relevant study in this area is the prediction work presented by Lui in (72), where the use of the K-Nearest-Neighbor algorithm is used to predict queue wait times. What we propose here is to use wait time predictions based on scheduler reservation tables (described later). The reservation is used by local scheduling policies to schedule jobs and to decide where and when the jobs will start. Thus, this prediction mechanism takes into account the current scheduling outcome, the status of the resources, and the scheduling policy used.

## 8.3 Contributions

In this Thesis we study the use of job-guided scheduling techniques in scenarios with distributed and heterogeneous resources. The main contributions of this work are:

- The scheduling policy is a job-guided policy. The users submit the job using the ISIS-Dispatcher (see figure 8.1). Each job is scheduled by an independent Dispatcher.
- The ISIS-Dispatcher is focused on optimizing the job metrics with the scheduling decisions, for instance the job average slowdown, the job response time, the wait time or the cost of the used resources assigned to the job.
- The application has not to be modified for use the proposed architecture. The scheduling is totally transparent to the user and the application.
- The scheduling is done according the information and functionalities that the local schedulers provide. Thus, there is an interaction between the two scheduling layers.
- We keep the local scheduling policies without important modifications. Centers do not have to adapt their scheduling policies or schedulers to ISIS-Dispatcher. They have to provide dynamic information about the system status, for example: the number of queued jobs, when a job would start if it were submitted to the center or which is the remaining computational work in the center.
- In the simulation environment used in the evaluation, we modeled the different levels of the scheduling architecture. Consequently, not only were the dispatcher

scheduling policy modeled, but also the local scheduling policies used in the experiments are modeled using an independent reservation table for each of them.

- We propose and evaluate the use of the Less-Waittime, Less-Waittime-pred, Less-Slowdown and Less-Cost-Sld task assignment policies that are based on the wait time and runtime predictions for the submitted jobs. The evaluation for this policy is compared with the task assignment policies described above, which can be applied in the job-guided scheduling ISIS-Dispatcher policy (SITA policies cannot).

## 8.4 The ISIS-Dispatcher

When the user wants to submit a job to the system, he/she contacts the ISIS-Dispatcher which manages his/her submissions and provides the static description of the job and the metric $\gamma$ to be optmized (i.e: the job wait time). In this evaluation, the user provides the script/executable file, the number of requested processors and the requested runtime time. Once the dispatcher has accepted the submission, it carries out the following steps:

1. For each computational resource $\sigma$ (with a particular configuration, particular characteristics, and managed by a given center) in all the resources available to the user:
   (a) The dispatcher checks that the static properties of $\sigma$ match the static description of the job. For example, it checks that the computational resource has equal or more processors than those requested by the job. [1]
   (b) In affirmative cases, the dispatcher contacts the predictor service and requests a prediction for job runtime in the given resource. $RTPred_{\alpha}(JobDescription, \sigma_i)$.
   (c) Once the dispatcher receives the job runtime prediction for the given job in the given resource it contacts the local scheduler to gather the required metric $\gamma$:
      i. It contacts the scheduler that manages the resource and requests the value of the metric: $\alpha_{\gamma,\sigma_i} = LocalModule.Perf(RTPred_{\alpha}, JobRequirements)$

[1]In architectures with thousands of resources, it is not feasible to contact all the resources. Future versions will use the information system and heuristics to decide which hosts the dispatcher must connect to, and which not.

ii. It adds the performance metric returned to the list of metrics ($metrics_{\{\alpha,\sigma_i\}}$) for the job in the given resource.

2. Given the complete list of retrieved metrics, $metrics_\alpha = \left\{\alpha_{\{\gamma,\sigma_1\}},..,\alpha_{\{\gamma,\sigma_n\}}\right\}$ , where a metric entry is composed of the metric value and the resource where the metric was requested. Using an optimization function, $SelectBestResource(metrics_\alpha)$, the best resource is selected based on the metrics that have been collected. For example, for the Less-WaitTime policy, the resource with the least predicted wait time is selected.
3. The dispatcher submits the job to the center that owns the resource.

## 8.5 The First Proposal: the Less-Waittime

In the initial proposal of the ISIS-Dispatcher strategy we evaluated the performance of the Less-Waittime task assignment policy against the policies that were proposed in the other works. In this first study we compared the four following different task assignment policies:

- The Less-JobWaittime policy minimizes the wait time for the job. Given a static description of a job, the local resource will provide the estimated wait time for the job based on the current resource state. We implement a prediction mechanism for different sets of scheduling policies (EASY-Backfilling, LXWF-Backfilling, Shorted Job Backfilled First and FCFS) that use a reservation table that simulates the possible scheduling outcome taking into account all the running and queued jobs at each point of time.
- The Less-JobsInQueue policy submits the job to the computational resource with the least number of queued jobs.
- The Less-Work-Left policy submits the job to the computational resource with the least amount of pending work.
- The Less-SubmittedJobs policy submits the jobs to the center with the least number of submitted jobs.

### 8.5.1 Job wait time prediction

The Less-JobWaittime task assignment policy submits the job to the center that returns the lowest predicted wait time. The approach taken in this evaluation was that each center has to provide such predictions. However, other architectures can also be used, for instance having several prediction/model services. In that case no interactions with the local centers would be required.

How to predict the amount of time that a given job will wait in the queue of a system has been explored by other researchers in several works (23)(89)(90)(72). What we propose in this Thesis is the use of reservation tables. The reservation is used by the local scheduling policies to schedule the jobs and decide where and when the jobs will start.

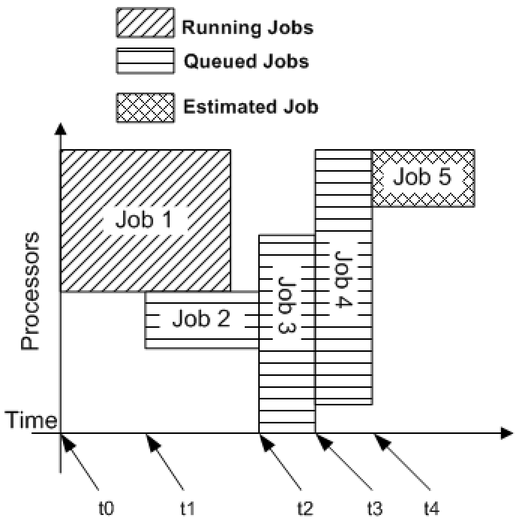
In this work the prediction mechanism uses the reservation table to estimate the possible scheduling outcome. It contains two different types of allocations: allocations for those jobs that are running; and pre-allocations for the queued jobs. The status of the reservation table in a given point of time is only one of all the possible scheduling outcomes and the current scheduling may change depending on the dynamicity of the scheduling policy. Also, the accuracy of the job runtime estimation or prediction has an impact on the dynamicity of the scheduling outcomes, mainly due to the job runtime overestimations.

The prediction of the wait time of a job at given point of time, that requires a given runtime and number of processors in a given resource, will be computed with: the earliest allocation that the job would receive in the resource given the current outcome if it was submitted at the provided time. Obviously, this allocation will depend on the scheduling policy used in the center, and probably will vary in different time stamps. All the scheduling events are reflected on the status of the reservation table. The prediction technique presented in this work is mainly designed for FCFS and backfilling policies.

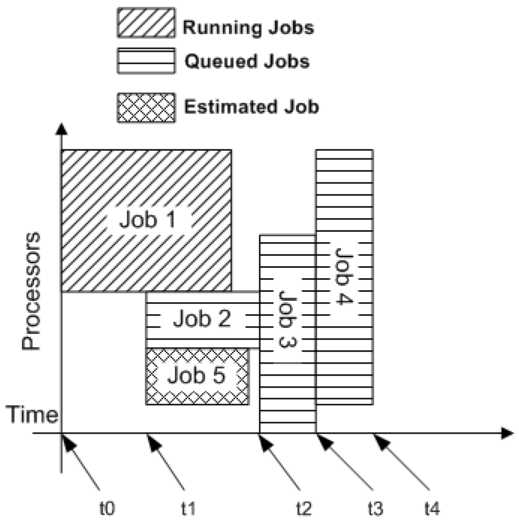
Figures 8.5 provide two examples of how a prediction for a new job would be computed in the two scheduling policies used in this chapter. In both examples the current time is $t_1$, there is one job running (*Job 1*), and three more queued (*Job 2*, *Job 3* and *Job 4*). If a prediction for the wait time for the job *Job 5* was required by a given instance of the ISIS-Dispatcher, the center would return $t_4 - t_1$ in the case of FCFS (Figure 8.5a) and would return 0 in the case of Backfilling (Figure 8.5b).

### 8.5.2 Evaluation

In this section we describe the simulation environment that was used to evaluate the presented policy and architecture. We characterize all the experiments, including the workloads and scenarios, that were designed to evaluate the current proposal.



(a) Prediction for FCFS



(b) Prediction for Backfilling

Figure 8.5: Wait time prediction

### Workloads

For the evaluation of the ISIS-Architecture we used as a source the following three workloads:

- The San Diego Supercomputer Center (SDSC) Blue Horizon log (144-node IBM SP, with 8 processors per node)
- The San Diego Supercomputer Center (SDSC-SP2) SP2 log (128-node IBM SP2)
- The Cornell Theory Center (CTC) SP2 log (34) (512-node IBM SP2).

For the simulation we used traces generated with the fusion of the first four months of each trace (FUSION). The following section describes the simulation: first we simulated the four months for each trace independently; second, using the unique fusion trace, different configurations of a distributed scenario composed by the three centers were simulated. We chose these workloads because they contain jobs with different levels of parallelism and with run times that vary substantially. More information about their properties and characteristics can be found in the workload archive of Dror Feitelson (29).

### Simulation Scenarios

In all the scenarios presented below, all the metrics presented in first part of this section were evaluated. In the second and third scenarios we also evaluated what happens when the characteristics of the underlying systems have different configurations, in terms of scheduling policies and computational resource configurations.

The characteristics of each of the evaluated scenarios are:

1. In the first scenario (ALL-SJBF), all the centers used the same policy: Shortest Job Backfilled First. The number of processors and computational resources were configured in exactly the same way as the original.

2. In the second scenario (CTC/4), the Shortest Job Backfilled First was also used for all the centers. However, in this case we emulated what would happen if the CTC center performed four times slower (having a *CPU Factor* of 4 [2]) than the two others . In this case, all the jobs that ran to this center spent four times longer than the runtime specified in the original workload. Although this approach is simple, it was enough to evaluate the impact of having resources with different computational power.

3. In the last scenario (CTC-FCFS), the SDSC and SDSC-Blue also used the Shortest Job Backfilled First policy. However, the CTC center used the FCFS scheduling policy. As in the first scenario, the computational resource configuration was exactly the same as the original.

The first scenario evaluates situations where all the hosts available have the same scheduling policy. Thus, each computational host is managed by the same scheduling policy and each computational unit (the processors) of all the hosts has the same computational power. Centers only differ in the number of processors. We defined the two other scenarios to evaluate how the ISIS-Dispatcher behaves with computational resources with different scheduling policies. In the second scenario we evaluated the impact of having heterogeneous resources. In this situation the CTC processors perform four times slower than the processors of the other two centers. In the last scenario we evaluated the impact of having different scheduling policies in the local hosts.

[2]When the job $\alpha$ with original runtime $\alpha_{\{OriginalRunTime,rt\}}$ (specified in the SWF trace used in the simulation) is submitted to the resource $\sigma$ the runtime of the job in the center becomes $\alpha_{\{RunTime,rt\}} = \alpha_{\{OriginalRunTime,rt\}} * CPUFactor$

| **Center** | **Estimator** | **BSLD** | **SLD** | **WaitTime** |
|---|---|---|---|---|
| SDSC | Mean | 8,66 | 12,9 | 2471 |
| | STDev | 47,7 | 86,07 | 8412 |
| | $95_{th}$ Percentile | 17,1 | 18,92 | 18101 |
| SDSC-Blue | Mean | 6,8 | 7,6 | 1331 |
| | STDev | 29 | 36 | 5207,2 |
| | $95_{th}$ Percentile | 28,5 | 29 | 8777 |
| CTC | Mean | 2,8 | 3,03 | 1182 |
| | STDev | 23 | 27,1 | 4307,3 |
| | $95_{th}$ Percentile | 2,3 | 2,5 | 6223 |
| CTC/4 | Mean | 19,8 | 20,467 | 9664 |
| | STDev | 57,23 | 58,203 | 20216 |
| | $95_{th}$ Percentile | 114,3 | 116,3 | 54450 |
| CTC FCFS | Mean | 12,833 | 14,04 | 3183,3 |
| | STDev | 66,54 | 77,03 | 9585 |
| | $95_{th}$ Percentile | 32,403 | 32,65 | 32996,4 |

Table 8.1: Performance Variables for each workload

### Performance results

Table 8.1 presents the performance metrics for the simulation of the workloads used in this evaluation (CTC, SDSC and SDSC-Blue) with SJ-Backfilled First in each center. We also include the simulations for the CTC with the other two different configurations that were used in the experiments of the distributed scenarios: the first includes the CTC, and the second also includes the CTC simulation but using the FCFS policy. As can be observed, the workload that has the best slowdown and wait time is the CTC. The SDSC and SDSC-Blue have a similar average bounded slowdown, however, the $95_{th}$ percentile of the SDSC-Blue is one order of magnitude greater than the SDSC. In terms of wait time, jobs remain longer in the wait queue in the workload of the SDSC than the other two. In terms of $95_{th}$ percentile the jobs spend three times longer in the SDSC than in the CTC.

The performance obtained when reducing the computational power and the policy of the CTC center (Table 8.1) is not surprising. Using FCFS or reducing by four the computational power of the CTC significantly increases the slowdown and wait time for the CTC workload. The capacity of the resource of execute the same workload was reduced four times. Thus, the original scenario cannot cope with the same job stream.

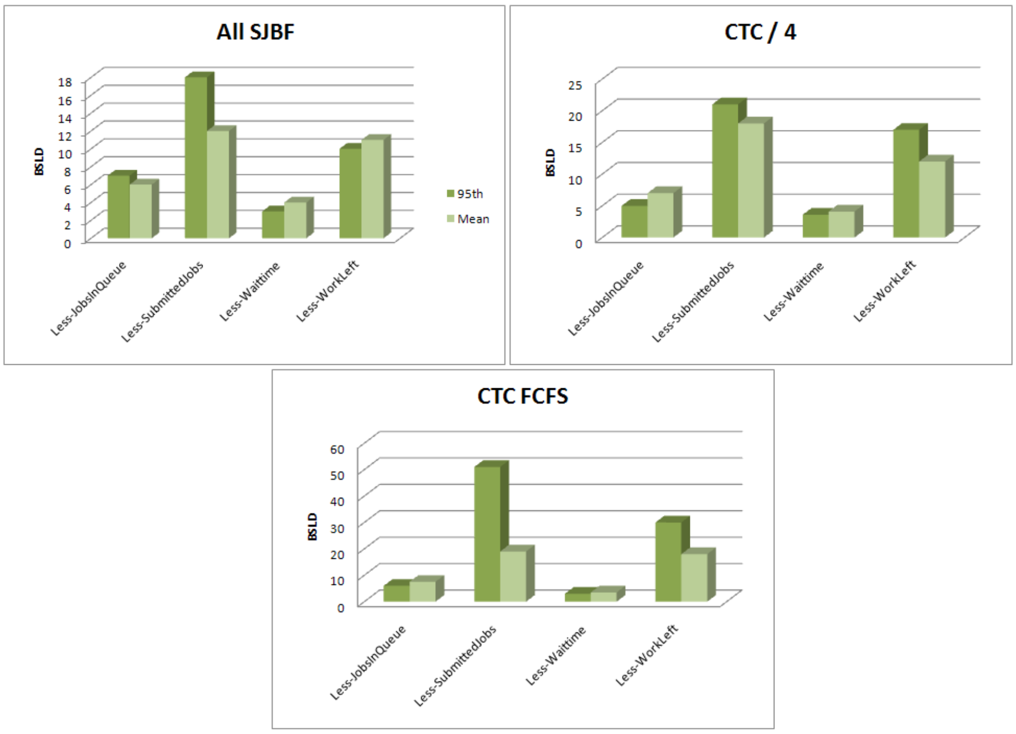


Figure 8.6: Bounded Slowdown

| Center | Estimator | Ratio BSLD | Ratio SLD | Ratio WaitTime |
|---|---|---|---|---|
| SDSC | Mean | 5,44 | 7,5 | 10,98 |
| | STDev | 5,1 | 9,27 | 4,36 |
| | $95_{th}$ Percentile | 14,1 | 14,92 | 53,2 |
| SDSC-Blue | Mean | 4,2 | 4,4 | 5,9 |
| | STDev | 3,1 | 3,6 | 2,7 |
| | $95_{th}$ Percentile | 23,5 | 22 | 25,8 |
| CTC | Mean | 1,8 | 1,7 | 5,2 |
| | STDev | 2,5 | 2,9,1 | 2,3 |
| | $95_{th}$ Percentile | 1,6 | 1,6 | 18 |

Table 8.2: Ratio: Original Job Perf. / ISIS Less-WaitTime Job Perf.

The main concern was to evaluate later this configuration in the distributed scenario.

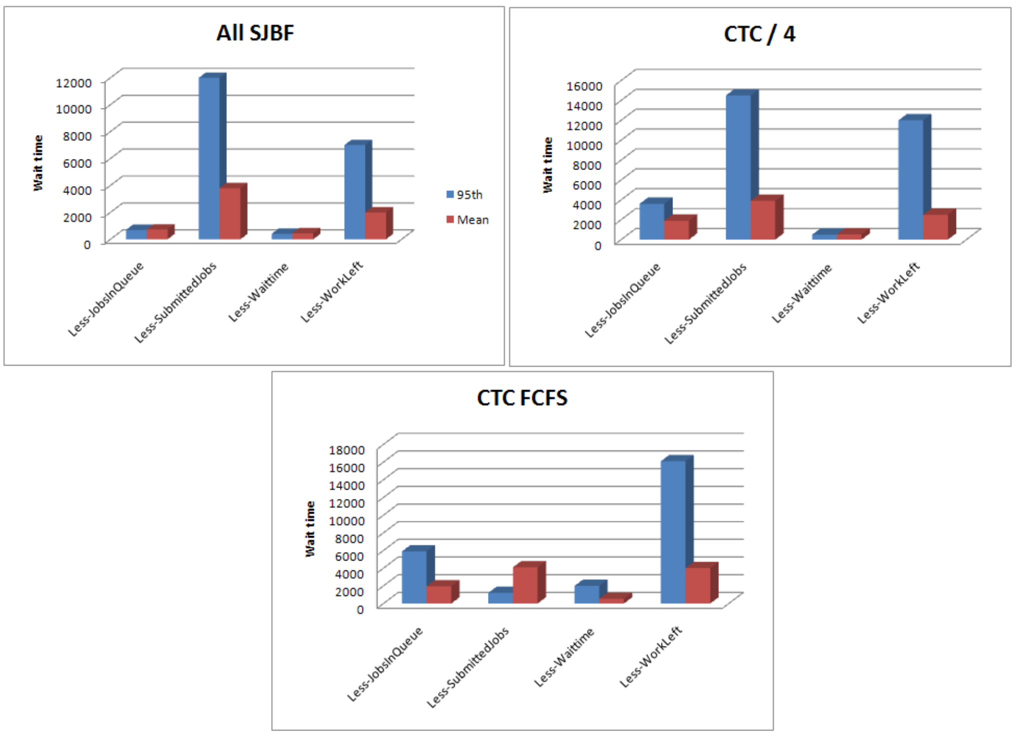


Figure 8.7: Wait time

Figure 8.6 presents the average (left column) and $95_{th}$ (rigth column) percentile for the bounded slowdown in the three presented scenarios and the different task assignment policies studied. In the *scenario 1* (all centers with SJ-Backfilled First), the Less-QueuedJobs and Less-WaitTime policies showed the best performance. However, the first one obtains a slowdown (1,7) twice as small as the second one (3,9). The other two policies performed substantially worst. The average slowdown and the $95_{th}$ percentile are three or even ten times greater than in the others. For instance the average slowdown of the Less-Waitime is around two while the same slowdown for the Less-Work-Left in the same scenario is around ten. The average wait time in this scenario (see Figure 8.7) presented similar behavior to the slowdown. However, the percentile shows that in the case of the Less-SubmittedJobs and Less-Work-Left the wait time of the jobs has a high variance. This fact is also corroborated by the standard deviation that the wait time experiments in both policies (see Figure 8.11).

The Less-WorkLeft policy takes into account the amount of pending work and the

Less-JobsInQueue not. Therefore, we expected that the first policy one would perform much better than the second one. However, the presented results showed the contrary. Analyzing both simulations we have stated that in some situations the Less-WorkLeft policy unbalances excessively the number of submitted jobs. As shown in table 8.3 it submits around 800 jobs more to the SDSC center than the Less-JobsInQueue. The figures 8.8 and 8.9 show that the amount of queued jobs in the SDSC is substantially bigger in the Less-WorkLeft policy in this specific interval of the simulation. This unbalance is caused by the characteristics of the stream of jobs that are submitted during this interval to the system. The initial part of this stream is composed by several jobs that requires from 256 processors until 800 processors and that have large runtime. Because of the capacity of the SDSC center (128 processors), these jobs can only be allocated to the CTC center (412 processors) and the SDSC-Blue (1152 processors). This causes that an important amount of smaller jobs (with less than 128 processors) have to be submitted to the SDSC center to accumulate the same amount of pending work that are assigned to the other two centers. Thus, as we stated in (55), this stream of jobs composed by jobs that requires all the processors of the host and jobs that requires small number of processors causes an important fragmentation in the scheduling of the SDSC. These situations occur several times in the simulation and they decrease substantially the performance achieved by the Less-WorkLeft policy.

| *Resource* | *Less-JobQueuedJobs* | *Less-WorkLeft* |
|---|---|---|
| CTC | 10788 | 10912 |
| SDSC | 1953 | 2560 |
| SDSC-Blue | 9550 | 8819 |

Table 8.3: Number of submitted jobs per host in Less-Jobs and Less-WorkLeft

What the results suggest is that the Less-SubmittedJobs policy has the worst performance of all the assignment policies, since the choice of where the job is submitted does not depend on the static properties of the job (estimated runtime and processors). Regarding the other two policies, the Less-JobsInQueue policy performs substantially better than the Less-WorkLeft.

Table 8.2 provides the ratio for the job performance variables in the original scenario (where jobs where submitted to the independent centers) against the performance for the jobs in the ALL-SJBF scenario. The results show that the jobs of all the centers obtained substantially better service in the new scenario. For instance, the average bounded slowdown in SDSC is 5.44 times greater than the average slowdown for the jobs in the *scenario 1*.

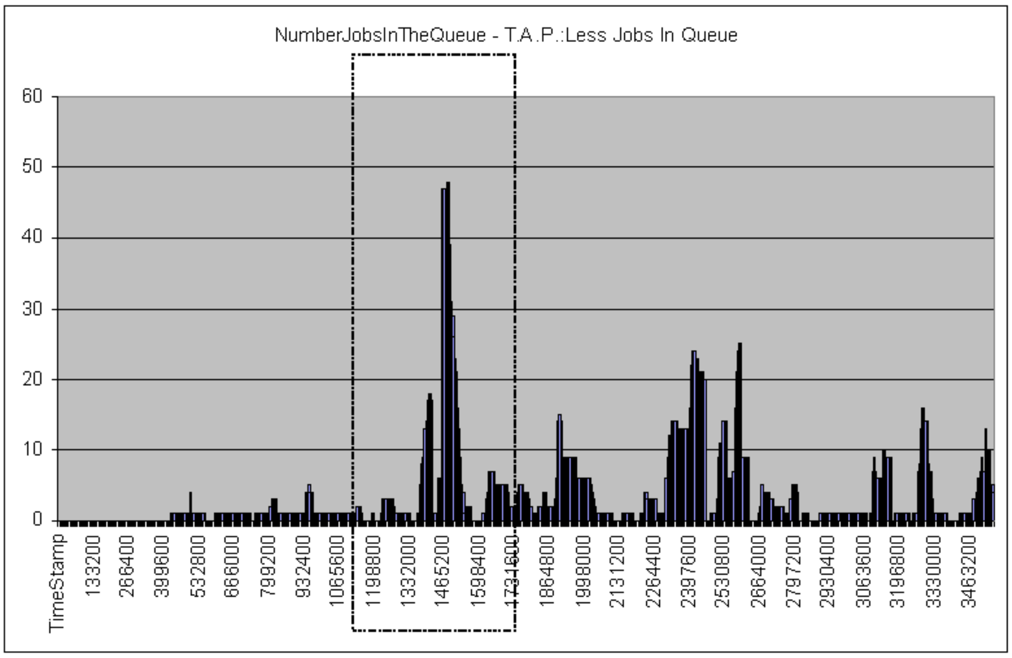


Figure 8.8: Number of queued jobs in the SDSC

The other two scenarios analyzed in the evaluation show that the ISIS-Dispacher scheduling policy is able to react in heterogeneous environments where the computational capabilities of the different centers can vary (in the *scenario 2* with a resource with less computational power and in the *scenario 3* with a resource with different scheduling policy). Compared to the *scenario 1* the performance shown in both scenarios experienced only a small drop. Thus, the system was able to schedule the jobs to the different resources adapting to the different capabilities of each of the available centers. This fact we observed in the simulations, is that in the situation where the CTC center used a scheduling policy with lower performance, the amount of workload was automatically balanced from this center to the SDSC-Blue and to the SDSC center (similar properties were found in the *scenario 3*). Regarding the performance achieved by each of the task assignment policies used in the experiments, the results show similar behaviors to those we observed in the first scenario.

Clearly, independently of the configuration used, using the Less-WaitTime assignment policy in the ISIS-Dispatcher scheduling policy obtained the best performance results in all the scenarios that were evaluated. It has demonstrated that it

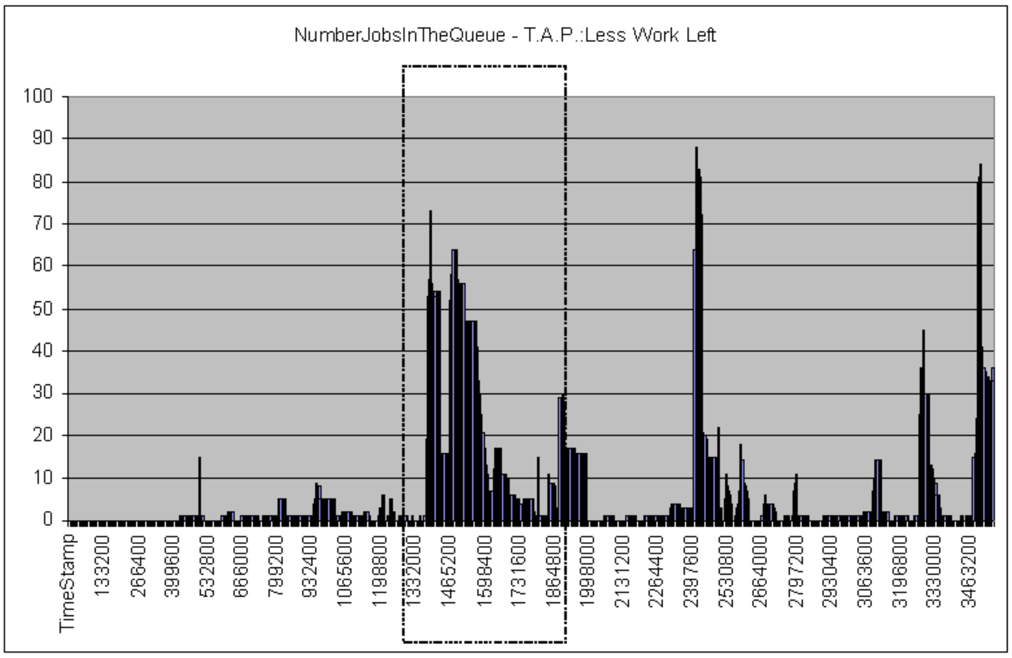


Figure 8.9: Number of queued jobs in the SDSC

is better able to adapt to the difference configuration of the local centers, and to provide a similar service to all the jobs.

The *scenarios 2 and 3* show that having a scheduling policy with a poor performance has more impact on the overall system than having a computational resource with a low computational power. Thus, improving the local scheduling policies can imply increasing the efficiency for the system more than replacing the existent hardware for a more powerful one. Consequently, in some situations investment on new computational resources can be saved by optimizing the policies.

## 8.6 The final proposal

In the previous subsection we presented the performance of Less-Waittime assignment policy that is based on the wait time prediction generated using the user runtime estimate. This analysis was carried out for evaluate the potential of this new approach in front of the already proposed techniques. Once, the benefits of our Thesis were stated we focused on designing more sophisticate approaches for include more capabilities into the existent

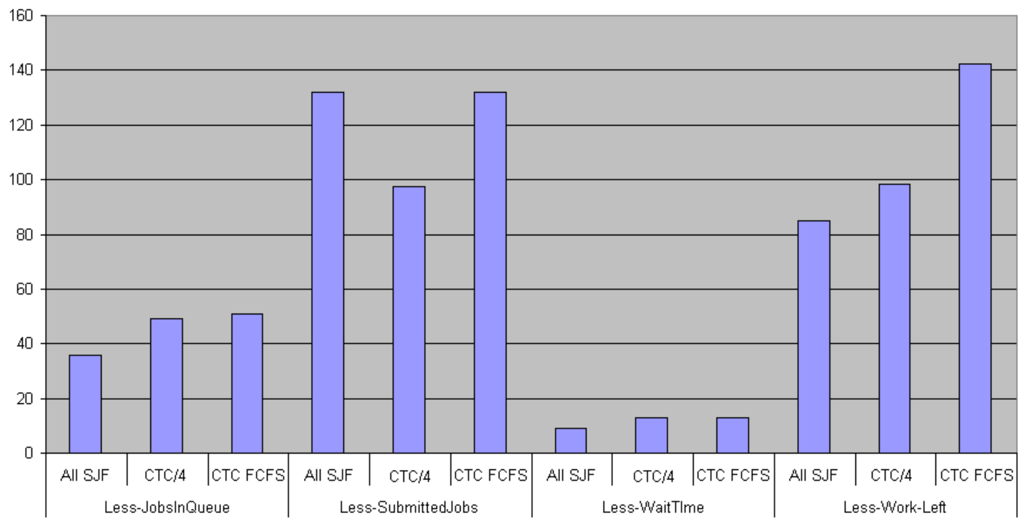


Figure 8.10: Bounded Slowdown Standard Deviation

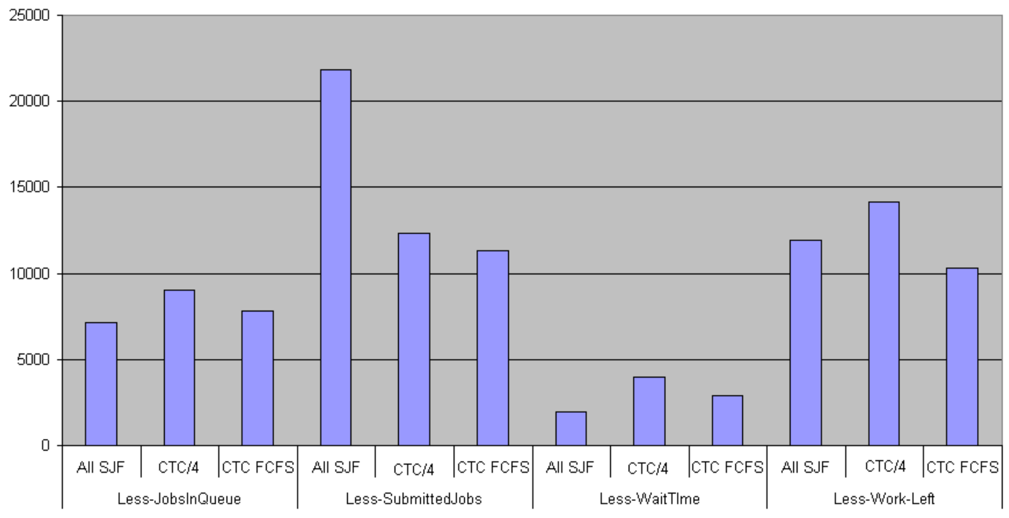


Figure 8.11: Wait time Standard Deviation

architecture.

In this section we present two new task assignment policies which use predicted job run and wait time and whose objective is to optimize slowdown and the economic cost of job execution. Furthermore, we propose a variant of Less-WaitTime which uses predicted runtime rather than user estimates. We describe a new evolutive prediction model based on classification trees and discretization techniques designed for these distributed environments. We introduce the modelling of the economic cost of using different computational resources. The evaluation of the new proposed techniques includes a study of the economic impact of each of the evaluation policies.

### 8.6.1 Job runtime prediction

Statistics provides many techniques for correlate and modelize variables, populations and distributions. This models are very suitable for carry out predictions for variables such as the runtime. However, some statistical algorithms require that all input variables be normal, in order to assure the correctness of the resulting conclusions. In usually this requires tedious analysis and in many situations this is a really complex task. We have already emphasized that data mining techniques (60) are especially interesting because they can be applied to many different kinds of data independently of their nature. Nevertheless, the most interesting characteristic of their algorithms is that some of them have an autonomous learning mechanism, and are able to derive or discover new knowledge without the necessary interaction of a third party (user, expert or other software component).

Several techniques can be used for performance predictions, for example Bayessian Networks (62), linear regressions (70)(95), C.45 and ID3 classification trees (9)(45), K-Means (8)(69) or X-Means (80). In our previous study we used C45 trees and discretization techniques to predict the job runtimes. In this section we present how we have constructed and validated the model that was used later in the dispatcher evaluation.

The prediction model that we have used in our experiments was developed using the Weka (63) framework. Its goal is to predict the runtime of a submitted job by using the static information provided by the user at submission time, and the historical information regarding the execution of finished jobs. The main idea is to classify job runtime according to a set of runtime intervals that have been defined based on historical information. Each of these intervals has an associated runtime prediction. Thus, when a job is classified in a given interval, the predictor returns the runtime associated with this interval. In this section we describe how the model is built during a simulation. We decided to use this classification model because in the study which we presented in (53) we stated that only very high quantitative errors had bad penalties in the performance of the scheduling policies. We present the first approximation about the presented model in

(59).

### Model generation

As we have already outlined above, the prediction model is composed of two different elements: the discretizer and the classifier. The first component is responsible for discretizing the job's continuous variables which are stored in the historical database to nominal values which are required in order to build the model. The second element, given a workload with the information (nominal variables) concerning the jobs which have already finished, allows the classification model to be constructed and carries out the job classification.

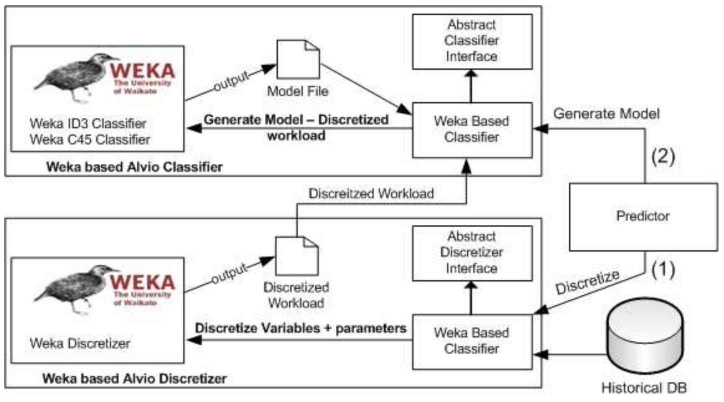

Abstract Classifier Interface
WEKA The University of Waikato
output
Model File
Weka ID3 Classifier
Weka C45 Classifier
Generate Model – Discretized workload
Weka Based Classifier
Generate Model
(2)
Weka based Alvio Classifier
Discreitzed Workload
Predictor
Abstract Discretizer Interface
WEKA The University of Waikato
output
Discretized Workload
Discretize
(1)
Weka Discretizer
Discretize Variables + parameters
Weka Based Classifier
Weka based Alvio Discretizer
Historical DB

Figure 8.12: Prediction Model Generation

The following steps are involved in the construction of the prediction model (see figure 8.12):

- Step 1. The job performance variables that are stored in the historical information database are discretized using the Weka discretizer. The number of intervals in which each of variables are discretized are computed by the Weka system, optimizing the number of bins using a leave-one-out estimate of the entropy (for equal-width binning) (39). In earlier versions of the prediction system, the number of intervals used to classify the continuous variables and the computational criteria were done manually. However, the resulting performance in the scheduling system and in prediction accuracy were substantially lower. The main reason for this is that the number of discretization intervals depends on the amount and characteristics of available historical information. Table 8.4 shows an example of the intervals that were defined in one of the model generations in one of the simulations presented in this chapter.

- Step 2. Using the workload generated in the previous step, the classification model is created using the ID3 Weka Classes functionalities. The input variables of the classification model that have to be generated are *JobNumber*, *JobSimSubmitTime*, *JobSimCenter*, *NumberProcessors*, *OriginalRequestedTime*, *RequestedMemory*, *UserID*, *GroupID* and *Executable*, and the response variable is the *runtime* of the job.

The input variables of the model generation are statically configured in the simulation. In our preliminary study, other variables, such as the *Average CPU Time Used* or the *Queue Number* were ruled out for two main reasons: the size of the pruned trees was big and the variables at submission time were not available. Using the discarded variables the model generation created trees with thousands of nodes with low recall and precision.

| | |
|---|---|
| JobNumber | (-inf-10.6],(10.6-20.2],(20.2-inf) |
| JobSimSubmitTime | (-inf-41901.3],(41901.3-83802.6],(83802.6-inf) |
| JobSimCenter | CTC,SDSC,SDSC-Blue |
| NumberProcessors | (-inf-103.3],(103.3-inf) |
| OriginalRequestedTime | (-inf-6750],(6750-13200],(13200-19650],(19650-inf) |
| RequestedMemory | All |
| UserID | (-inf-10.6],(10.6-20.2],(20.2-29.8],(29.8-inf) |
| GroupID | All |
| Executable | (-inf-29.8],(29.8-inf) |
| Runtime | (-inf-9289.8],(9289.8-18520.6], (18520.6-27751.4],(27751.4-36982.2], (36982.2-46213],(46213-inf) |

Table 8.4: Job Variables Discretization

The prediction model has to be regenerated each *K* job finish in order to include the new information about the exececution of the jobs that have finished from the last model generation. In the simulation experiments (see below) where we tested different configurations of this *K* value, we empirically determined that updating the model each 100 job finalizations provides accurate enough predictions.

### Prediction generation

In the previous subsection we described how the prediction model is constructed. When a job has to be scheduled, the dispatcher asks the prediction system for an estimate of its runtime, providing the statical description provided by the user. The variables that the user has to provide are the variables that have been used as input variables for the model construction (*JobNumber*, *JobSimSubmitTime* etc). Given the job $\alpha$ description the prediction system does the following:

- Firstly, each job variable value is discretized based on the discretization intervals resulting from the model creation ( step *(1)* - Figure 8.13). For example, given intervals defined in the example of the table 8.4 and the job definition $\alpha = \{JobNumber = 2, OriginalRequestedTime = 232...\}$ the resulting discretization would be:
  $\alpha = \{JobNumber = (-inf - 10.6], OriginalRequestedTime = (-inf - 6750]...\}$

- Secondly, the discretized values of the job description are provided to the classifier. Using the classification model previously generated and the values, the job is classified in one of the runtime intervals computed in the model generation.

- Finally, a continuous value is returned, based on the interval returned. This value is the upper bound of the interval if the interval returned is the one of the *N-1* intervals, for the last interval the lower bound is returned.

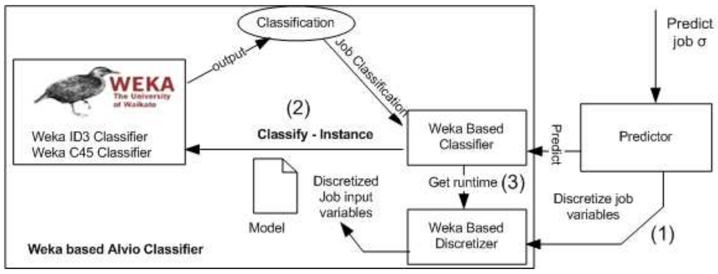
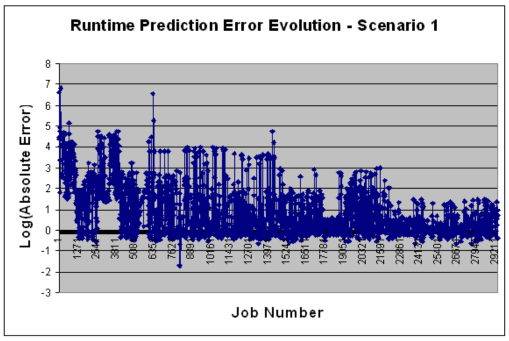


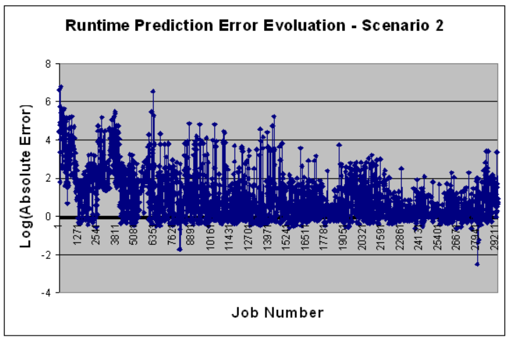
Figure 8.13: Job Prediction

## 8.6.2 Prediction Evaluation

Table 8.5 shows the *median*, the *IQR* and $95_{th}Percentile$ for the percentage of prediction errors of the wait time (using user estimates and the prediction system in the model), the job runtime prediction and the slowdown prediction for the two evaluated scenarios. The variables are not grouped by experiments because the error predictions shown in the simulation were very similar in the experiments in the same scenario.



(a) Scenario 1



(b) Scenario 2

Figure 8.14: Prediction Error Evolution

The evaluation of the runtime prediction model presented in 8.6.1 shows that the accuracy of the returned predictions improves substantially over time. Figure 8.14 shows the evolution of runtime error prediction during the simulation. It shows the logarithmic scale of error prediction for each of the executed jobs in each of the two scenarios described in the preceding section. As can be observed in both scenarios, the error decreases six times from the initial predictions, and the job runtime predictions of the last submitted jobs show a clear decrease in errors. The model shows that the more job executions information feedback it has, the more it is able to learn and carry out better classification, and, as a result, it has a warm up interval where the returned predictions are inaccurate. The model behaves slightly worse in the second scenario, this is probably due to the entropy added by having different scheduling policies in the underlying systems.

The wait time prediction model that we introduced did not show the same error prediction tendency. On the contrary, it provided more accurate predictions from initial jobs submissions. Furthermore, the errors followed a similar pattern during the whole system simulation, with all the predictions very close to the average values shown in the table 8.5. This prediction mechanism is based on the job runtime estimation provided by the dispatcher to the local scheduler, and thus its accuracy is affected by the accuracy of these estimations. In our experiments we evaluated how the wait time prediction is affected by user estimation and by the prediction service estimation. As can be observed

| **Scenario** | Variable | $95_{th} Percentile$ | **Median** | **IQR** |
|---|---|---|---|---|
| Scenario 1 | WT(UEst) | 98.04 | 12.34 | 52.79 |
| | WT(Pred) | 117.76 | 18.56 | 262.70 |
| | RT Prediction | 3020 | 11.81 | 145.47 |
| | SLD Prediction | 9145.9 | 121.22 | 130.52 |
| Scenario 2 | WT(UEst) | 927.63 | 97.20 | 255.31 |
| | WT(Pred) | 1173.76 | 51.56 | 262.70 |
| | RT Prediction | 4685 | 88.33 | 796.44 |
| | SLD Prediction | 8941.53 | 100 | 144.38 |

Table 8.5: Prediction Accuracy - Percentage

in the table 8.5 the wait time prediction shows similar average errors in the first scenario using user estimates or the prediction service. However, the results obtained from the second scenario are substantially better using the prediction service.

The computation of the predicted slowdown uses the predicted wait time and the predicted runtime. Hence, its accuracy showed a similar pattern to that shown in the runtime predictions. In the warm up period, the runtime prediction of the slowdown was very inaccurate, but it improved during the prediction model evolution. The predicted slowdown in both scenarios showed average errors around the 100%, but showed more accurate predictions (average errors around 30%) in the last 40% of estimated jobs.

### 8.6.3 The economic model

As will be discussed in the following subsection, using the assignment policy slowdown with economic considerations, a user can tell the dispatcher the maximum amount of money that he/she wants to spend on job execution. To evaluate this capability we generated an extended SWF trace that contained the maximum amount of money that the user wanted to spend on each job. To generate these traces, and to compute the cost associated with executing a job in one of the three centers, we used the following considerations [3]:

- In the simulation scenarios that were evaluated in the experiments, the computational powers associated with the processors are different (different

[3]The formulas and values that we used to compute the economic cost of the use of each resource are a simplification of the formulas that are used in the real research centers such as the CERN Computing Units (76) or the Allocation Units used in the European Projects

*CPUFactors*). Thus, as is specified in the table 8.6 the processors for the SDSC center performed three times faster that the processors of the CTC and four times faster that the processors of the SDSC-Blue.

- The computational power that the center $\sigma$ has is computed as $\sigma_{power} = \sigma_{processors} * \sigma_{CPUFactor}$.
- The execution of a job in the center $\sigma$ has a cost of *N Allocations Units* (AU). The allocation units that we associate with a given center is computed as $(\frac{\sigma_{power}}{\gamma}) * 100$ where $\gamma$ is the minimum of the computational powers of all the centers. Thus the allocation units per hour are proportional to the computational power available in the center (see table 8.6). We multiply the fraction by 100 because we want to have an economic cost closer to the real cost charged in real HPC projects.
- The economic cost associated with using a given computational resource was configured manually. Usually the cost of an allocation unit of a given center takes into account the cost of maintenance of the resource, administrators, electricity and so on. Because of this we decided to set the cost proportionally to the size of the center (see table 8.6).

Taking the previous characterizations into account, the economic cost of executing a job $\alpha$ in the center $\sigma$ is computed as:
$cost = \sigma_{AU/h} * \sigma_{Cost(AU)} * \alpha_{cpus} * \alpha_{RT}$, where the job runtime is specified in hours and the number of allocations unit per hour and cost of allocations unit depends on the center.

| **Center** | **CPU Factor** | **Power** | **AU/H** | **AU Cost** |
|---|---|---|---|---|
| SDSC | 0.5 | $64 = 128_{cpus}/0.5$ | 100 | 0.5 euros |
| CTC | 1.5 | $274.6 = 412_{cpus}/1.5$ | 420 | 1.12 euros |
| SDSC-Blue | 2 | $64 = 1152_{cpus}/2$ | 900 | 2 euros |

Table 8.6: Economic Costs Definition

The maximum *desired* costs for each job included in the simulation *FUSSION* trace was computed using the previous formula and: the estimated job runtime that the user provided in the original trace; and the AU/h and cost AU of the center where they were originally executed with the original workload.

### 8.6.4 The ISIS-Dispatcher task assignment policies

For the final ISIS architecture that we propose in this Thesis we include four different task assignment policies (note that the first one was described and evaluated in the section 8.5):

- The Less-WaitTime (LWT) policy minimizes wait time for the job. Given a static description of a job, including the user runtime estimation, the local resource provides the estimated wait time for the job based on the current resource state. As described in the initial part of this chapter, the prediction mechanism uses a reservation table which simulates the possible scheduling outcome, taking into account all the running and queued jobs at each point of time.
- The Less-WaitTime-pred (LWT-pred) policy minimizes wait time for the job using the predicted runtime provided by the prediction system.
- The Less-Slowdown(LSLD) policy minimizes slowdown for the job. Given the static description of the job, the dispatcher estimates the slowdown of the job in a given resource center. To do this, it asks the prediction system for the job runtime prediction and then asks the local resource for an estimate of job wait time in this resource.
- The Less-Cost-Sld (LCOST) minimizes job slowdown taking into account the economic cost of using each of the local resources. Like the two previous policies it uses prediction techniques to estimate job runtime and job wait time.

In the rest of this section we introduce in more detail the two new policies which we propose to use in the context of the ISIS-Dispatcher system. In the first experiments with this strategy we evaluated different task assignment policies which other authors proposed in the literature and the Less-WaitTime policy (LWT). We stated how the use of prediction techniques for waiting time used in the new Less-WaitTime policy can substantially improve the overall performance with regard to other policies. In this section we will compare the three new policies with the Less-WaitTime policy.

### 8.6.5 The Less-Slowdown policy

The objective of this policy is to optimize slowdown for the job $\alpha$ which the dispatcher is scheduling. As already stated above, slowdown for a job in a given resource $\sigma$ is computed as follows:

1. The dispatcher asks the local scheduler that manages the resource for an estimate of the wait time for the job in the current scheduling $WT_{(\alpha,\sigma)}$.
2. The dispatcher asks the prediction service for an estimate of the job runtime in the resource $RT_{(\alpha,\sigma)}$.

3. The estimated slowdown for the job is computed using the two previous estimations. $SLD_{(\alpha,\sigma)} = \frac{RT_{(\alpha,\sigma)}+WT_{(\alpha,\sigma)}}{RT_{(\alpha,\sigma)}}$

The selection policy will chose the computational resource with the least estimated slowdown.

### 8.6.6 The Less-Slowdown policy with economic considerations

In our research projects, the computational resources assign an economic cost per hour of job execution to the users and to the projects which use them. Many researchers have studied the economic considerations in the scheduling in multi-site architectures. All of this studies have proposed economic models for centralized approaches (27)(12)(78). Probably, one of the most reference work is the Buyya Thesis that is oriented to provide resource management and scheduling techniques for Grid Computing based on economic considerations (10)(11). In this section we present a economic based task assignment policy derived from the Less-Slowdown.

The objective of this policy is to take into account these costs when selecting the resource where the job will be submitted. The main goal of this metric is again to optimize slowdown for the job, but a secondary goal is to optimize the cost of executing the job. Using this metric the user can provide a maximum cost that he/she would like to spend on the job execution ($\alpha_{MaxAllowedCost}$). It works as follows:

- Like the Less-Slowdown policy, for each of the available computational resources, it computes the runtime prediction $RT_{(\alpha,\sigma)}$, the wait time estimate $WT_{(\alpha,\sigma)}$ and the estimated slowdown $SLD_{(\alpha,\sigma)}$.
- All the resources which do not satisfy the $CostPerHour_{\sigma} * RT_{(\alpha,\sigma)} <= \alpha_{MaxAllowedCost}$ are discarded. In the case that there is no resource that satisfies this condition, the resource with the least economic cost is selected.
- Given all the resources $\sigma_1,..,\sigma_n$ that satisfy the previous restriction, using the strict weak ordering less than comparison function presented in the figure 8.15, the center that satisfies the minimum function is selected. [4]

[4]Note that, as is defined in the STL (50), this Strict Weak Ordering is a Binary Predicate which compares two possible submissions, returning true if the first precedes the second. This predicate must satisfy the standard mathematical definition of a strict weak ordering. The precise requirements are stated below, but what they roughly mean is that a Strict Weak Ordering has to behave the way that ”less than” behaves: if a is less than b then b is not less than a, if a is less than b and b is less than c then a is less than c, and so on.

$$factor_{sld} = \frac{EstSld(\alpha,\sigma_1)}{EstSld(\alpha,\sigma_2)} \tag{8.1}$$

$$factor_{cost} = \frac{EstCost(\alpha,\sigma_2)}{EstCost(\alpha,\sigma_1)} \tag{8.2}$$

$$better(\sigma_1,\sigma_2) = (\frac{factor_{sld}}{factor_{cost}} > 1) \tag{8.3}$$

Figure 8.15: Strict Weak Ordering Function

Given the two centers $\sigma_1$ and $\sigma_2$, the previous comparison function determines if, in the case that the dispatcher submits the job to the second center, the job would receive a better service in terms of slowdown and economic cost. The first part of the formula *(1)* computes the improvement in terms of slowdown which the job would receive if it were submitted in the center $\sigma_2$ compared to the first center. The second part of the formula *(2)* computes the increase in the economic cost of executing the job in the center $\sigma_2$ compared to the cost of the first center. If the improvement in the slowdown of submitting the job to center $\sigma_1$ rather than $\sigma_2$ is proportionate or better to the increase in the cost of executing the job in the second center, then the function will return true. The main goal of this function is to determine if the increased cost of using a *better* computational resource is more than compensated for by an improvement of service received by the job.

Note that in this evaluation we consider that the $\alpha_{MaxAllowedCost}$ is a soft requirement where the user can specify an approximate limit on the money that the execution of the job will cost. However, as we are using prediction techniques, this limit might be violated. For example, if a job is submitted to the most expensive center because the prediction service estimates that a given job will take 120 minutes, and in fact the job takes 420 minutes, the final cost of the job execution will be several times more than the user intended. In cases where the user has a hard requirement concerning the $\alpha_{MaxAllowedCost}$ in the real system we should not use the prediction service, but rather we should use the estimated runtime provided by the user. However, this last situation can lead to very inaccurate estimates as the user does not know which resource will finally handle the job.

### 8.6.7 Architecture Evaluation

In all the scenarios presented below, all the policies presented in section 8.6.4 were evaluated. The computational power of each of the resources was fixed in all scenarios. In the simulation models we configured the *CPU Factor* (see subsection 8.6.3) of the SDSC center as *0.5*, the *CPU Factor* of the CTC as *1.5* and the *CPU Factor* of the center SDSC-Blue as *2*. What is different in each scenario is the scheduling policy used by the local schedulers. The characteristics of each of the evaluated scenarios are as follows:

1. In the first scenario (*Scenario 1*), all the centers used the same policy: Shortest Job Backfilled First .

2. In the second scenario (*Scenario 2*), the SDSC scheduler used the EASY-Backfilling scheduling policy, SDSC-Blue used the Shortest Job Backfilled First policy, and finally the CTC center used the FCFS scheduling policy.

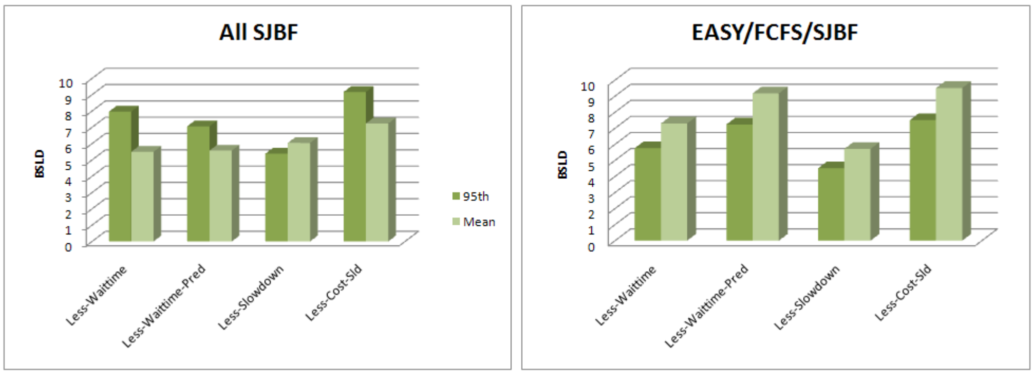
The first scenario evaluates situations where all available hosts have the same scheduling policy. In the second, more plausible, scenario we evaluated the impact of having different scheduling policies with different characteristics and performance. The main goal of these scenarios was to evaluate the influence of error prediction on different policies, and to examine how these differences impact on the decision of each of the metrics which the ISIS-Dispatcher uses.

The experiments consisted of simulating the *FUSSION* trace (described in subsection 8.5.2) in two scenarios (*Scenario 1* and *Scenario 2*) and using the four metrics described in section 8.6.4 (Less-JobWaitTime, Less-WaitTime-pred, Less-Slowdown and Less-Cost-Sld)

#### Performance results

The figures 8.16, 8.17, 8.18 and 8.19 show the performance variables for each experiment that was evaluated in this study, arranged by scenario and the metric used. The figures show the $95_{th}$ *percentile* and the mean for the bounded slowdown (8.16) and the wait time (8.17), the economic cost of executing the whole workload (8.19) (see the cost computation of the job execution in subsection 8.6.3), the $95_{th}$ *Percentile* of the cost of executing one job in the system (8.18), and finally, the $95_{th}$ *Percentile* of the number of used processors (8.19).

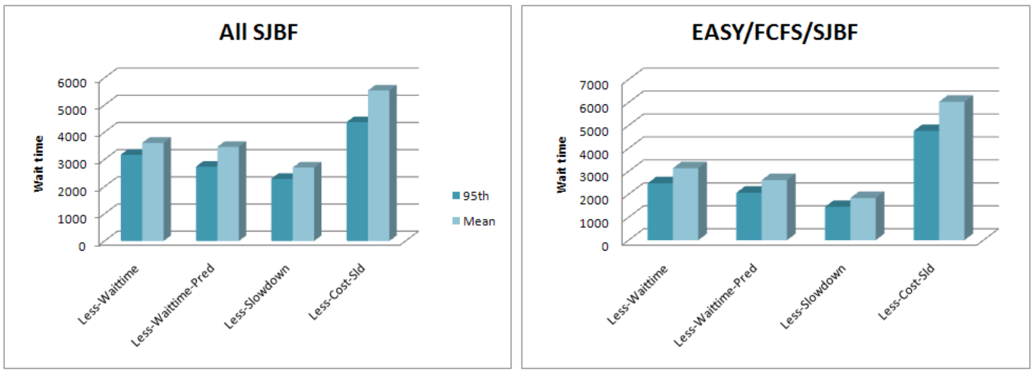
The LWT-pred policy demonstrated that using prediction techniques to estimate job runtime and job wait time allows the dispatcher to achieve similar or even better performance results than using the user estimates provided by the user. In both scenarios



(a) Scenario 1 (b) Scenario 2

Figure 8.16: BoundedSlowdown

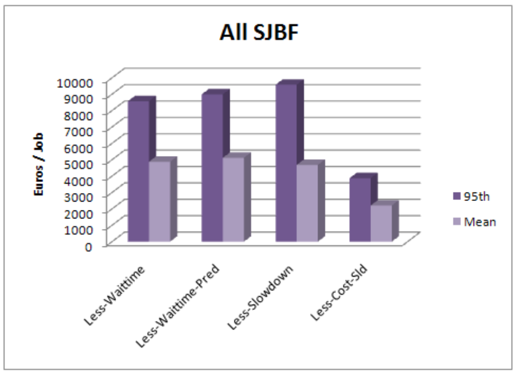
the Less-WT showed slightly better $95_{th}Perc$ of the wait time that the LWT-pred. However, the LWT-pred showed that job slowdown is better optimized by using the prediction service than by using user estimates. As can be observed in figure 8.19, in both scenarios with the two policies the utilization of the processors available in the system showed similar results.



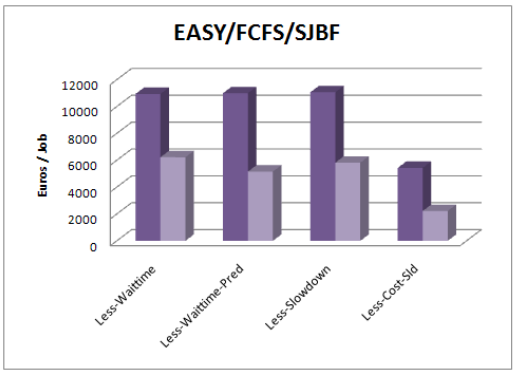
(a) Scenario 1 (b) Scenario 2

Figure 8.17: Wait time

Results showed that the LSLD optimizes job slowdown more efficiently than the Less-WT and LWT-pred policies. The difference is higher in the second scenario, where this policy achieved a $95_{th}Perc$ of the slowdown 8 percent, and 10 percent lower than the other two policies respectively. However, this policy not only improves the slowdown compared to the other two, it also improves the $95_{th}Perc$ of the wait time in 200 seconds and 1000 seconds in the first and second scenario the $95_{th}Perc$ achieved by the LWT-pred. Furthermore the utilization of the resources of the system were improved, the utilizations of the processors of the systems is improved in both scenarios. The weak point that we detected in the LSLD policy compared to the LWTs policies is the economic cost of job execution. Using this first selection policy the whole execution of the workload has an increment related to the job execution 14% and 5% compared to the LWT and LWT-pred, in the first scenario, and of 18% and 7% compared to the same policies in the second scenario.



(a) Scenario 1

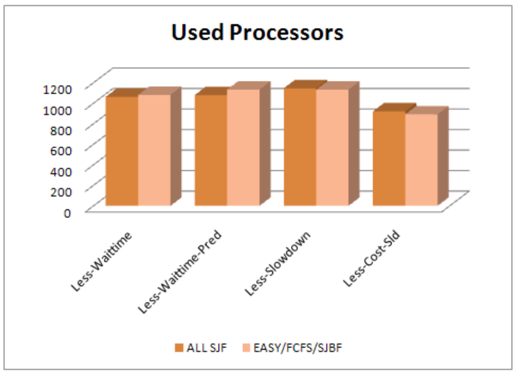


(b) Scenario 2

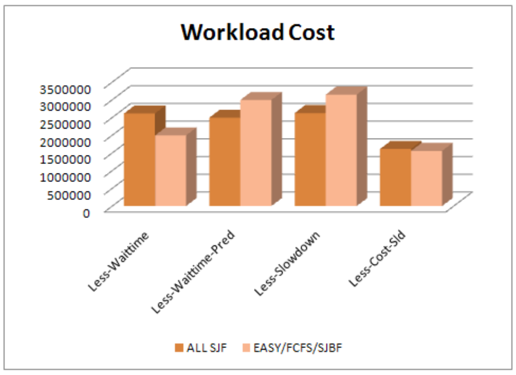
Figure 8.18: Cost in euros/job

Finally, the LCOST policy clearly optimizes the economic cost of job execution in both scenarios. For example, the execution of the whole workload for the first scenario is around 50% cheaper than the economic cost of the execution of the other three workloads. Furthermore, the $95_{th}Perc$ cost associated with the individual execution of a job was reduced by 5000 euros with respect to the three initial task assignment policies. The other positive fact is that the number of fullfilled agreements in the number of maximum desired cost increased over 90% of all jobs in both scenarios.

However, this reduction of economic cost was accompanied by an increase in slowdown and wait times for scheduled jobs. As can be observed in figures 8.16 and 8.17 the $95_{th}Perc$ increased by 20% compared to the other three metrics. Similarly, as can be observed in the number of processors used on average, overall utilization of the resources of the system fell. Thus, although the economic cost to the individual user



(a) Used processors



(b) Workload Cost

Figure 8.19: Global Variables

decreased, so too did utilization of the resources, the performance of the system.

The increase in slowdown and the reduction in resource utilization is due to two main factors. Firstly, predicting small slowdown (seconds) of a job in a given center when it is really much longer (minutes) causes bad scheduling decisions. The difference in the computational cost of resources makes users select the cheaper resource unless the slowdown difference is really high. In the initial phases of the prediction model, the slowdown prediction is inaccurate because the runtime predicted is relatively small (a few minutes) compared to the real runtime of all the large jobs. In these cases the slowdown factor described in 8.6.6 is not high enough and the cheaper resource is selected. The second issue is that the computational cost that we assigned to the different resources is really high, thus the chance of the SDSC-Blue resource being used is substantially lower than the two others.

## 8.7 Summary

In this chapter we have presented the use of a job-guided scheduling technique designed to be deployed in multi-site architectures composed of a set of computational resources with different architecture configurations and different scheduling policies. The main goal of the technique presented here is to provide the user with a scheduling entity that will decide the most appropriate computational resource to submit his/her jobs to. The interesting property of the ISIS-Dispatcher policy is that it can be applied to heterogeneous systems, i.e. it does not presuppose any of the properties or capabilities of a particular system.

The scheduling policy presented here uses a set of task assignment policies to decide where the jobs are finally submitted. We have proposed three new task assignment policies that use predicted job run and wait time and whose objective is to optimize waittime (Less-WT and Less-WT-pred), slowdown (Less-Slowdown) and the economic cost associated with execution, but taking into account slowdown (Less-Cost-Sld).

The evaluation of all the architecture has been done in two steps. First, we have presented the evaluation about how the most representative task assignment scheduling policies presented in the literature perform in the policy presented here (including the Less-WorkLeft, Less-Less-SubmittedJobs and Less-JobsInQueue policies). We have compared this results with the Less-WT policy (that uses the runtime estimate) proposed in this Thesis. We have evaluated the proposal in three different scenarios using the workloads CTC, SDSC and SDSC-Blue from the Workload Log Archive. The first scenario was composed of a set of centers with the same scheduling policies and different computational resources (different numbers of processors); the second was composed of a set of centers with different scheduling policies (two with Shorted Job Backfilled First and one with FCFS) and different computational resources ; and finally, the last one was composed of centers with the same policies and different computational resources with different computational power (one of the centers performed four times slower than the other two). Although the scheduling proposal of our Thesis is non-centralized and the dispatcher does not store any information regarding the progress of the global scheduling, it has been shown that using the appropriate task assignment policy (in this analysis the Less-Waittime policy showed the most promising results) it is able to achieve good global performance, adapting to the underlying center resource characteristics and to the local scheduling policies.

In the second evaluation we have outlined and evaluated the use of prediction techniques in the ISIS-Dispatcher architecture and compared them to the use of user estimates that has been presented in the first evaluation. We have presented the evaluation of the rest of the ISIS-Dispatcher task assignment policies. We have also described a new evolutive prediction model based on classification trees and discretization techniques designed for these distributed environments. We have also explained how we introduced modelling of the economic cost of the use of different computational resources into the architecture. Evaluations of the new proposed techniques include a study of the economic impact of each of the evaluation policies.

The results obtained in the simulation experiments evaluated in this second evaluation show how the use of prediction techniques can actually allow the Less-WaitTime policy to achieve better results than the use of user estimates. Users can submit their applications without providing job runtime estimates, which can be complicated in these scenarios. We have also shown how the prediction model described in this chapter can provide accurate job runtime predictions and how it is able to learn and improve during the life cycle of the global system. Thus, each time more historical information is available, the prediction service is able to make more accurate predictions with more sophisticated internal structures.

With reference to the new task assignment policies proposed in this second evaluation, we have demonstrated how the performance of the system can be improved by trying to minimize job slowdown, using job run and wait time predictions. The results have shown how the Less-Slowdown assignment policy substantially reduces bounded slowdown and wait time of the system, while it increases utilization of resources. In this new version of the ISIS-Dispatcher, an economic model has been introduced and the cost of using each computational resource has been modeled and considered as part of the analysis. The Less-Cost-Sld task assignment policy demostrated that it was able to improve the economic cost related to job execution reducing by up to 50% the cost of executing the whole workload. However, slowdown increased by around 20% and wait time also increased. On the other hand, although the economic cost of individual jobs falls, utilization of computational resources is also reduced. We claim that this problem is mainly caused by using slowdown predictions of very small jobs (i.e. 4.3) in a given center, while the real jobs are in fact much longer (i.e: 78.2). This causes bad scheduling decisions, since the difference in computational cost of resources makes users choose the cheaper resource unless the slowdown difference is really high.

# Chapter 9

# Conclusions

In this Thesis we have proposed strategies to improve the performance of the local scheduling scenarios and the meta-scheduling scenarios. In the first one, we have been focused in providing mechanism for evaluate prediction based backfilling scheduling policies and improve the performance of the local scenarios by considering how the shared resources are used by the running jobs. In the meta-scheduler scenario we have been focused in providing a non-centralized scheduling strategies for large multi-site scenarios composed by different local-scheduling scenarios.

In the first part of this Thesis, we have described and evaluated a set of f-model based prediction models that characterize the behavior that prediction techniques have shown in HPC centers. They have been designed to evaluate scheduling policies that use predictions rather than user estimates. We have formalized two different type of prediction error models: quantitative errors and qualitative errors. The first group consists on returning as a prediction the real runtime of the job plus a percentage of error. We have defined a set of variants of this model based on to which kind of jobs the quantitative error has to be applied (i.e: to the jobs with large runtime or jobs that use large number of processors). The second group of presented prediction error models consists on changing the nature of the job with the prediction, for instance predicting the job as short while it is really large. We have provided an evaluation of the impact of these prediction models to a set of backfilling variants. The analysis have stated that when the errors are applied randomly to the 50% of all the stream of jobs the scheduling performance is clearly affected by high quantitative errors (more than 5000%). Another interesting remark has been that the negative errors have not resulted in important impact on the performance of the system as we had expected. Surprisingly, if the quantitative errors are applied to a determined subset of jobs (large or short number of processors etc.) based on the required resources there is no relevant impact in the performance of the system. However, when this quantitative prediction errors have been applied to the prediction of the short jobs the performance of the average bounded bounded slowdown experiments a smoothly increment until errors of 400%, and from this point it experiments a exponential increment. Taking into account the results obtained in this study, we support that it is not necessary to design specialized predictors for determined job types based on their resource requirements for achieve good performance. However, predictors should have to be as much accurate as possible for those jobs that are likely to be short, and try to avoid high quantitative errors on them. On the other hand, the qualitative errors have to be avoided when predicting the job runtime for the short and large jobs. These errors have clear exponential tendency on the average bounded slowdown for all the evaluated workloads when the number of jobs being miss categorized is incremented.

We have also evaluated the impact of considering the penalty introduced in the job runtime due to resource sharing in system performance metrics in the backfilling policies. To do this, we have formalized a model that characterize how the resources are being shared by the allocated jobs. This model punishes the run time for the running jobs that are sharing a given resource in those cases that the resource demand is higher than the resource capacity. This evaluation has been done using the Alvio simulator. Using this model, we have evaluated the impact of the memory bandwidth in the Shortest Job Backfilled first backfilling variant with two well-known workloads. The results have shown quantitative differences between the performance obtained using the presented model or not. In both workloads, incrementing the number of jobs with high memory bandwidth demand has resulted in an increment of all the performance metrics. In all the experiments, the performance metrics have shown a clear increment in all the experiments. Furthermore, the number of killed jobs has experimented an evident increment.

Using this model we have designed two new Resource Selection Policies and one new Job Scheduling Policy. First, the *Find Less Consume Distribution* that attempts to minimize the job runtime penalty that an allocated job will experience. Based on the utilization status of the shared resources in current scheduling outcome and job resource requirements, the LessConsume policy allocates each job process to the free allocations in which the job is expected to experience the lowest penalties. Second, we have also described the *Find Less Consume Threshold Distribution* selection policy which finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. This last Resource Selection Policy has been designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job.

We have evaluate the performance of these strategies using the same workloads that were used to evaluate the resource usage model. We have evaluated the two resource selection policies (the LessConsumeThreshold with Thresholds *1*, *1.15*, *1.25* and *1.5*) with the Shortest Job Backfilled first. Both resource selection policies show how the performance of the system can be improved by considering where the jobs are finally allocated. The bounded slowdown of both policies show slightly higher values than those achieved by a First Fit resource selection policy. However, they show a very important improvement in the percentage of penalized runtime of jobs, and more importantly, in the number of killed jobs, showing a very good balance in the increment of the BSLD. Both have reduced until the 16% if the number of killed jobs in all the evaluated workloads respect the backfilling with First Fit resource selection policy scneario.

Using the Less Consume Threshold Resource Selection Policy, we have proposed a new backfilling strategy : the *Resource Usage Aware Backfilling* (RUA-Backfilling) job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different *Threshold* configurations.

The evaluation of the RUA-Backfilling has demonstrated how the exchange of scheduling information between the Local Resource Manager and the Scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close start time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time of the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with LessConsume scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

In the second part of the Thesis, focused on meta-scheduling scenarios, we have proposed the usage of self-scheduling techniques for dispatching the jobs that are submitted to a set of distributed computational hosts that are managed by independent schedulers (such as MOAB or LoadLeveler). To do that, we have designed the ISIS-Dispatcher that is a per-job-scheduler that is associated to one and only one job. Therefore, once the user wants to submit a job, a new dispatcher is instantiated. It is responsible for submitting the job to the computational resource that best satisfies the job requirements.

The ISIS-Dispatcher has been designed for deployment in large systems, for instance groups of HPC Research Centers or groups of universities. The core of the ISIS-Dispatcher algorithm is based on task selection policies. We have proposed four new task assignment policies that use the predicted job run and wait time [1] and whom objective is to optimize the job wait time (Less-WT and Less-WT-pred), the job slowdown (Less-SLD-pred) and the economic cost of the job execution (Less-SLD-pred-econ) of the job respectively. In the presented work we have proposed and evaluate the usage of prediction techniques in rather than the user estimates. We have described a new evolutive prediction model based on classification trees and discretization techniques designed for these distributed environments.

In the evaluation of this proposal, we have introduced the modelization of the economic cost associate to the usage of the different computational resources. Thus the evaluations of the new proposed techniques include an study of the economic impact of each of the evaluated policies. The architecture has been evaluated with workload created containing the execution of three different well-known workloads of a set of HPC Centers. Although the scheduling proposal is non-centralized and the dispatcher does not store any information regarding the progress of the global scheduling, it has been shown that using the appropriate task assignment policy it is able to achieve good global performance, adapting to the underlying center resource characteristics and to the local scheduling policies and improving also the job packing. The results obtained in the simulation experiments show how the use of prediction techniques can actually allow the policy to achieve better results than the use of user estimates. Users can submit their applications without providing job runtime estimates, which can be complicated in these scenarios. We have also shown how the prediction model can provide accurate job runtime predictions and how it is able to learn and improve during the life cycle of the global system. Thus, each time more historical information is available, the prediction service is able to make more accurate predictions with more sophisticated internal structures.

Regarding the new task selection policies, the results have shown how the Less-WT and Less-WT-pred task assignment policies are able to reduce the amount of wait time of the scheduled jobs four times respect the other proposals. The second one, corroborates how the usage of prediction techniques rather than user estimates can provide substantially benefits to the system performance providing higher performance results than the first one. On the other hand the Less-SLD-pred assignment policy substantially reduces bounded slowdown and wait time of the system, while it increases utilization of resources. The Less-SDL-pred-econ task assignment policy demostrated that it was able to improve the economic cost related to job execution reducing by up

[1]This work has been based on the work done in the usage of prediction techniques in local scenarios.

to 50% the cost of executing the whole workload. On the other hand, although the economic cost of individual jobs falls, utilization of computational resources is also reduced. We claim that this problem is mainly caused by using slowdown predictions of very small jobs (i.e. 4.3) in a given center, while the real jobs are in fact much longer (i.e: 78.2). This causes bad scheduling decisions, since the difference in computational cost of resources makes users choose the cheaper resource unless the slowdown difference is really high.

In the final part of the Thesis we have presented the Grid Backfilling meta-scheduling policy. We have shown how a coordinated scheduling among all the different centers using data mining prediction techniques can substantially improve the performance of the global distributed infrastructure, and can provide a uniform access to the user to all the heterogeneous Grid resources. We describe this meta-scheduling policy that optimizes the global utilization of the system resources and decreases substantially the response time for the jobs. We also present how data mining techniques applied to historical information can provide very suitable inputs for carrying out the Grid Backfilling meta-scheduling decisions.

The evaluation of this policy has been tested using a model that characterizes the computational resources for NGS architecture and a workload of four years of this Grid. This architecture includes the centers of Oxford, Manchester, LR and Leeds, and has simulated the Grid Backfilling policy. The results have shown that a global scheduling carried out in top of all the centers improves qualitatively the service provided and reduces substantially the response time for the submitted jobs. The average wait time of all the centers has been reduced qualitatively. For instance the Manchester average wait is almost two times bigger than the average wait time experimented in the Grid Backfilling. Furthermore, the average wait time of the Oxford center is around forty times bigger than the Grid Backfilling once. The number of backfilled jobs per day has shown how the global backfilling approach give more chances to the jobs to start earlier rather than using independent scheduling per centers. This variable shows how the ratio of backfilled jobs is at minimum two times bigger than the once achieved in the original configurations.

## 9.1 Future Work

In the local scenario we have presented two different strategies for improve the performance and service of the local scheduling architectures. Firstly, we have presented a set of models that allow to evaluate backfilling variants that use prediction rather than user estimates. They have provided strong results for support the usage of prediction techniques rather than user estimates. As a part of the future work is important to evaluate the usage of specific prediction systems in real environments. Thus, we will consider to extend some of our scheduling architectures for usage such techniques. Related to this, we should have to carry out studies about how the prediction techniques, including the new scheduling interfaces (such as the Deadline miss managers), can be included in the comercial Schedulers such as MOAB or LoadLeveler.

Secondly, we have presented a model for considering the resource sharing usage in the evaluation of job scheduling strategies. Together to this model, we have proposed two new resource selection policies and one job scheduling policy designed to minimize the resource sharing saturation and consider how the jobs are allocated to the processors of the architecture. In this Thesis we have evaluated the impact of the memory bandwidth usage in the backfilling scheduling policies, and we have compared its results with the performance obtained by the strategies proposed in the Thesis. Future research must include studies that consider how other shared resources may impact in the performance of the system. Clearly, the penalty function that has been presented in our model, has to be extended for consider penalties that other typologies of resource may show. For instance, the network bandwidth shows patterns in the job execution that are not considered in the penalty function described in this Thesis. Related to this, the results of our Thesis have demonstrated that there is important impact when considering the resource sharing, however the penalty models must be validated with real executions or specialized simulators. To achieve this goal, we could do this using the Dimemas (66) simulator or use other analytic tools such as paraver (81). In the evaluations that have been presented we have supposed that we have a perfect estimation about the memory bandwidth that the jobs will use during its runtime. The future research work have to evaluate the impact of having inaccurate estimations in the job resource sharing requirements. Similar to the usage of prediction techniques, the set of scheduling strategies proposed in this Thesis should be implemented in real systems and see how the model behaves with real workloads.

In the meta-scheduling scenario we have introduced the ISIS-Dispatcher scheduling approach, a non-centralized scheduling policy oriented to improve job performance variables. In the Thesis have deeply described its architectures and a set of new task assignment policies that have been designed for be used for this scheduling entity.

All of these strategies are based in the job wait and run time predictions. In this Thesis we have introduced a evolutive model based on data mining prediction techniques for the runtime estimation. Note, that in the evaluation of the ISIS we have emulated the effect of considering heterogeneous architectures (in terms of scheduling capabilities and hardware characteristics). However, real systems may show more dynamic behavior that may have more effects in the prediction accuracy. On the other hand, future work must evaluate the effect of using another wait time prediction technique based on the usage of a prediction service rather than querying the local schedulers. We can not expect that all the local systems available to the user will be able to provide the wait time prediction or include the technique proposed in our Thesis.

In the proposal of the ISIS-Dispatcher architecture that we have evaluated in this Thesis, we have considered that the dispatcher contact to all the local schedulers to gather the required information. In large scenarios, composed by many of these local scenarios, it is not feasible to make the dispatcher contact to all the systems. This may imply problems of scalability and the local schedulers may become saturated for the amount of queries. For face this challenge, we will have to study mechanisms about how the dispatcher gathers the information (i.e: using a information system that gathers all the information from the centers) and about who gathers the information (i.e: only carrying out queries to the centers to whom the user job has more affinity or randomly asking to a fixed number of centers of the system).

The decision about where the job has to be submitted has been based in the optimization of a given job metric. For instance, we have evaluated the performance of the job and the system when the dispatcher uses the Less-WT-pred metric (optimizing the wait time for the job). Thus, the center that has provided the best metric value has been selected for submit it. However, in many situations different users may have different goals. Thereby, future extensions and evaluations of the dispatching algorithm should model that different users may have different goals and may require to optimize different metrics. Related to this optimization, futures versions will have to consider to optimize a set of metrics rather than only one.

# Bibliography

[1] K. Aida, H. Kasahara, and S. Narita. Job scheduling scheme for pure space sharing among rigid jobs. pages 98–121, 1998. Lect. Notes Comput. Sci. vol. 1459.

[2] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[3] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting multiple levels of parallelism in openmp: A case study. In *International Conference on Parallel Processing*, pages 172–180, 1999.

[4] N. Bansal and M. Harchol-Balter. *Analysis of SRPT scheduling: investigating unfairness*. 2001.

[5] F. Berman and R. Wolski. Scheduling from the perspective of the application. pages 100–111, 1996.

[6] F. Berman and R. Wolski. The apples project: A status report. 1997.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[8] L. Bottou and Y. Bengio. Convergence properties of the $K$-means algorithms. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 585–592. The MIT Press, 1995.

[9] W. Buntine. Learning classification trees. In D. J. Hand, editor, *Artificial Intelligence frontiers in statistics*, pages 182–201. Chapman & Hall,London, 1993.

[10] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing, 2002.

[11] R. Buyya, D. Abramson, and S. Venugopal. The grid economy, 2004.

[12] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proceedings of the SPIE International Conference on Commercial Applications for High-Performance Computing*, Denver, USA, August 20-24 2001.

[13] M. Calzarossa, G. Haring, G. Kotsis, A. Merlo, and D. Tessera. A hierarchical approach to workload characterization for parallel systems. *Performance Computing and Networking, Lect. Notes Comput. Sci. vol.*, page pp. 102109, 1995.

[14] M. Calzarossa, L.Massari, , and D. Tessera. Workload characterization issues and methodologies. *In Performance Evaluation: Origins and Directions, Lect. Notes Comput. Sci.*, page pp. 459482, 2000.

[15] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proc. IEEE*, 81:1136–1150, 1993.

[16] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. *Job Scheduling Strategies for Parallel Processing*, vol 1659:pp. 66–89, 1999.

[17] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. *8th International Workshop on Job Scheduling Strategies for Parallel Processing*, Vol. 2537:103 – 127, 2002.

[18] W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. *4th Ann. Workshop Workload Characterization*, 2001.

[19] W. Cirne and F. Berman. A model for moldable supercomputer jobs. *15th Intl. Parallel and Distributed Processing Symp.*, 2001.

[20] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage : A measurement based study on unix. *IEEE Tans. Sotfw. Eng.*, pages 15(12) and pp. 1579–1586, 1989.

[21] P. Dinda. Online prediction of the running time of tasks. *Cluster Computing SIGMETRICS/Performance*, pages 225–236, 2002.

[22] A. B. Downey. A parallel workload model and its implications for processor allocation. *6th Intl. Symp. High Performance Distributed Comput.*, Aug 1997.

[23] A. B. Downey. Using queue time predictions for processor allocation. *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes In Computer Science; Vol. 1291:35 – 57, 1997.

[24] S. P. Ellner. Review of R, version 1.1.1. *Bulletin of the Ecological Society of America*, 82(2):127–128, April 2001.

[25] T. Elomaa. In defense of c4.5: Notes on learning one-level decision trees. pages 62–69, 1994.

[26] C. Ernemann, V. Hamscher, , and R. Yahyapour. Benefits of global grid computing for job scheduling. *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[27] C. Ernemann, V. Hamscher, and R. Yahyapour. Economic scheduling in grid computing, 2002.

[28] D. D. G. Feitelson. Parallel workload archive, 2007.

[29] D. D. G. Feitelson. Standard workload format, 2007.

[30] D. G. Feitelson. Packing schemes for gang scheduling'. *Job Scheduling Strategies for Parallel Processing*, Lect. Notes Comput. Sci. 1162:pp. 89–110, 1996.

[31] D. G. Feitelson. Workload modeling for performance evaluation. *In Performance Evaluation of Complex Systems: Techniques and Tools, Lect. Notes Comput. Sci. vol. 2459*, pages pp. 114–141, 2002.

[32] D. G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. *In Job Scheduling Strategies for Parallel Processing, Lect. Notes Comput. Sci.*, vol. 949:pp. 337–360, 1995.

[33] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. *In Job Scheduling Strategies for Parallel Processing, Lect. Notes Comput. Sci.*, vol. 1162:pp. 337–360, 1996.

[34] D. G. Feitelson and L. Rudolph. Workload evolution on the cornell theory center ibm sp2. *Job Scheduling Strategies for Parallel Processing*, 1162:pp. 27–40, 1996.

[35] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. *In Job Scheduling Strategies for Parallel Processing, Lect. Notes Comput. Sci.*, vol. 1459:pp. 1–24, 1998.

[36] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004*, 3277 / 2005:9, June 2004.

[37] D. G. Feitelson and D. Tsafrir. Workload sanitation for performance evaluation. *In IEEE Intl. Symp. Performance Analysis of Systems and Software*, pages pp. 221–230, 2006.

[38] F.Guim and J. Corbalan. A job self-scheduling policy for hpc infrastructures. *Job Scheduling Strategies for Parallel Processing: 13th International Workshop, JSSPP 2007*, 2007.

[39] J. Fisher and J. Principe. A methodology for information theoretic feature extraction, 1998.

[40] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *J Intl - International Journal of Supercomputer Applications.*, 1997.

[41] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[42] T. A. S. Foundation. Xalan xslt support engine. Website, 2007.

[43] T. A. S. Foundation. Xerces xml parser engine. Website, 2007.

[44] P. L. Geissler, C. Dellago, D. Chandler, J. Hutter, and M. Parrinello. Ab initio analysis of proton transfer dynamics, 2000.

[45] D. Geman and B. Jedynak. Model-based classification trees, 2001.

[46] S. Gerald, K. Rajkumar, R. Arun, and S. Ponnuswamy. Scheduling of parallel jobs in a heterogeneous multi-site environment. *JSSPP 2003*, 2003.

[47] R. Gibbons. A historical application profiler for use by parallel schedulers. *Job Scheduling Strategies for Parallel Processing 1997*, 1997.

[48] A. Goyenechea, F. Guim, I. Rodero, G. Terstyansky, and J. Corbalan. Extracting performance hints for grid users using data mining techniques: a case study in the ngs. *Mediterranean Journal: Special issue on data mining*, 2006.

[49] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[50] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic programming and high-performance libraries. *Int. J. Parallel Program.*, 33(2):145–164, 2005.

[51] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. (CS-94-21), 8, 1994.

[52] W. Gropp and E. Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):103–114, 1997.

[53] F. Guim and J. Corbalan. Prediction f based models for evaluating backfilling scheduling policies. *The 8th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2007.

[54] F. Guim, J. Corbalan, and J. Labarta. Analyzing loadleveler historical information for performance prediction. *Jornadas de Paralelismo 2005 and Granada*, 2005.

[55] F. Guim, J. Corbalan, and J. Labarta. Modeling the impact of resource sharing in backfilling policies using the alvio simulator. *15th Annual Meeting of the IEEE / ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007.

[56] F. Guim, J. Corbalan, and I. Rodero. The alvio simulator site, 2007.

[57] F. Guim, J. Corbalan, and I. Rodero. The alvio simulator tutorial, 2007.

[58] F. Guim, A. Goyeneche, J. Corbalan, J. Labarta, and G. Terstyansky. Grid computing performance prediction based in historical information. *Integrated Research in Grid Computing. November 2005. CoreGRID Integration Workshop*, 2005.

[59] F. Guim, A. Goyeneche, I. Rodero, and J. Corbalan. The grid backfilling: a multi-site scheduling architecture with data mining prediction techniques. *Second CoreGRID Workshop on Middleware at ISC2007*, 2007.

[60] J. Han and M. Kamber. Book: Data mining: Concepts and techniques. *Book*, 2001.

[61] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing*, 59(2):204–228, 1999.

[62] D. Heckerman. A tutorial on learning with bayesian networks, 1995.

[63] G. Holmes, A. Donkin, , and I. Witten. Weka: A machine learning workbench. *In Proc Second Australia and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia*, 1994.

[64] J. Hursey, E. Mallove, J. M. Squyres, and A. Lumsdaine. An extensible framework for distributed testing of mpi implementations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[65] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[66] T. C. Jess Labarta, Sergi Girona. Analyzing scheduling policies using dimemas. *3rd Workshop on environment and tools for parallel scientific computation. Parallel Computing*, 1997.

[67] V. L. K. Windisch, R. Moore, D. Feitelson, , and B. Nitzberg. A comparison of workload traces from two production parallel machines. *In 6th Symp. Frontiers Massively Parallel Comput.*, pages pp.319–326, 1996.

[68] S. Kannan. Workload management with loadleveler, 2007.

[69] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: analysis and implementation, 2002.

[70] A. Karalic. Employing linear regression in regression tree leaves. In *European Conference on Artificial Intelligence*, pages 440–441, 1992.

[71] K. Kurowski, A. Oleksiak, J. Nabrzyski, A. Kwiecien, M. Wojtkiewicz, M. Dyvzkowski, F. Guim, J. Corbalan, and J. Labarta. Multi-criteria grid resource management using performance prediction techniques. *Integrated Research in Grid Computing. November 2005. CoreGRID Integration Workshop*, 2005.

[72] H. Li, J. Chen, Y. Tao, D. Groep, , and L. Wolters. Improving a local learning technique for queue wait time predictions. *Cluster and Grid computing*, 2006.

[73] LLNL. Slurm local resource manager - site, 2007.

[74] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *J. Parallel and Distributed Computing*, 11:pp. 1105–1122, November 2003.

[75] T. MathWorks. Matlab mathematical software. Website, 2007.

[76] E. McIntosh. Benchmarking computers for hep. Technical report, CERN, Geneva, Switzerland, 1992.

[77] Minitab. Minitab statistical software. Website, 2007.

[78] R. Moreno and A. Alonso-Conde. Job scheduling and resource management techniques in economic grid environments, 2003.

[79] M. Neumüller and J. N. Wilson. Compact in-memory representation of XML data. Technical report, Department of Computer and Information Science, University of Strathclyde, Glasgow, Scotland, UK, 2002.

[80] D. Pelleg and A. Moore. $X$-means: Extending $K$-means with efficient estimation of the number of clusters. In *Proc. 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[81] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. 44:17–31, 1995.

[82] C. Pinchak, P. Lu, and M. Goldenberg. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. *Job Scheduling Strategies for Parallel Processing*, pages 205–228, 2002. Lect. Notes Comput. Sci. vol. 2537.

[83] C. Resources. Maui local scheduler - site, 2007.

[84] C. Resources. Moab local scheduler - site, 2007.

[85] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. pages 231–244, Jan. 2002.

[86] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *Cluster Computing 2004*, 2004.

[87] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, pages pp. 107–140, 1994.

[88] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The easy - loadleveler api project. *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes In Computer Science; Vol. 1162 archive:41 – 47, 1996.

[89] W. Smith, V. E. Taylor, and I. T. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. *Proceedings of the Job*

*Scheduling Strategies for Parallel Processing*, Lecture Notes In Computer Science; Vol. 1659:202 – 219, 1999.

[90] W. Smith and P. Wong. Resource selection using execution and queue wait time. predictions. page 7.

[91] D. Talby and D. Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. *Parallel Processing Symposium*, pages pp. 513–517, 1999.

[92] D. Talby, D. Tsafrir, Z. Goldberg, , and D. G. Feitelson. Session-based and estimation-less and and information-less runtime prediction algorithms for parallel and grid job scheduling. Technical report, School of Computer Science and Engineering and The Hebrew University of Jerusalem, 2006.

[93] I. The Wikimedia Foundation. Wikipedia. Website, 2001.

[94] J. D. F. Theodore Wilbur Anderson. *The New Statistical Analysis of Data*. Springer, 1996.

[95] H. Toutenburg. Prediction of response values in linear regression models from replicated experiments, 1998.

[96] D. Tsafrir, Y. Etsion, , and D. G. Feitelson. Modeling user runtime estimates. *In the 11th Workshop on Job Scheduling Strategies for Parallel Processing,Lecture Notes in Computer Science*, Vol.3834:pp. 1–35, 2006.

[97] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using runtime predictions rather than user estimates. *Technical Report 2005-5, School of Computer Science and Engineering, The Hebrew University of Jerusalem.*, 2005.

[98] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *In the IEEE TPDS*, 2006.

[99] D. Tsafrir and D. G. Feitelson. Workload flurries. Technical report, School of Computer Science and Engineering and The Hebrew University of Jerusalem, 2003.

[100] D. Tsafrir and D. G. Feitelson. Instability in parallel job scheduling simulation: the role of workload flurries. *In 20th Intl. Parallel and Distributed Processing Symp*, 2006.

[101] P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.

[102] T. H. y. R. J. W. Wonnacott. *ntroduccin a la estadstica*. Limusa/IPN, 1998.

[103] J. Yue. Global backfilling scheduling in multiclusters. *Asian Applied Computing Conference, AACC 2004*, pages pp. 232–239, 2004.

[104] Y. Zhang, W. Sun, , and Y. Inoguchi. Cpu load predictions on the computational grid. *Cluster and Grid computing*, 2006.