Hanzich M, Hernández P, Giné F *et al.* On/off-line prediction applied to job scheduling on non-dedicated NOWs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(1): 99–116 Jan. 2011. DOI 10.1007/s11390-011-1114-y

On/Off-Line Prediction Applied to Job Scheduling on Non-Dedicated NOWs

Mauricio Hanzich[1], Porfidio Hernández[2], Francesc Giné[3], Francesc Solsona[3], and Josep L. Lérida[3]

[1]*Department of Computer Application in Science and Engineering, Barcelona Supercomputing Center, Barcelona, Spain*

[2]*Department of Computer Architecture & Operating Systems, Autonomous University of Barcelona, Barcelona, Spain*

[3]*Department of Computer Science, University of Lleida, Lleida, Spain*

E-mail: mauricio.hanzich@bsc.es; porfidio.hernandez@uab.cat; {sisco,francesc,jlerida}@diei.udl.cat

Received December 23, 2009; revised November 19, 2010.

Abstract This paper proposes a prediction engine designed for non-dedicated clusters, which is able to estimate the turnaround time for parallel applications, even in the presence of serial workload of the workstation owner. The prediction engine can be configured to work with three different estimation kernels: a *Historical* kernel, a *Simulation* kernel based on analytical models and an integration of both, named *Hybrid* kernel. These estimation proposals were integrated into a scheduling system, named CISNE, which can be executed in an on-line or off-line mode. The accuracy of the proposed estimation methods was evaluated in relation to different job scheduling policies in a real and a simulated cluster environment. In both environments, we observed that the *Hybrid* system gives the best results because it combines the ability of a simulation engine to capture the dynamism of a non-dedicated environment together with the accuracy of the historical methods to estimate the application runtime considering the state of the resources.

Keywords prediction method, non-dedicated cluster, cluster computing, job scheduling, simulation

1 Introduction

Several studies[1-3] have revealed that a high percentage of the computing resources in a Network Of Workstations (NOW/Cluster) is idle. NOWs are owned by many universities, business and industry institutions. They are made of not necessarily the fastest computers and networks, but still form a huge computational power that can be used to solve many problems that require parallel computing for high performance[4]. To do this, they can be used as dedicated clusters during weekends and at nights or as *non-dedicated cluster* during working hours. In a non-dedicated cluster, individual PCs are used by their local users to run sequential application (local jobs), while the cluster as a whole or its subset could be employed by cluster users to run parallel applications. The possibility of executing parallel jobs without perturbing the performance of the local user applications on each workstation has led to a proposal for new job schedulers[5-9].

In a NOW environment, when a parallel job is submitted to the job scheduler, it waits in a queue until it is scheduled and executed. Thus, the scheduler must deal with the job selection problem from the waiting queue, together with the problem of selecting the best set of nodes for executing a job. In order to solve both problems in the best way, the scheduler should be able to estimate the future cluster state (available resources, number of idle nodes and intrusion level into the load workload)[10-11]. Based on these predictions, the scheduler can select the best parallel job from the waiting queue to be launched to the system, or choose the combination of resources from the available resource pool that is expected to maximize performance for a given application. Thus, performance prediction turns into a critical component of scheduling environments.

These considerations have stimulated some studies into performance estimation in parallel environments. These can be classified into three different approaches. The most widely used is based on a *Historical system* that records the past executions of an application[11-16]. The second alternative is to use a *Simulation system based on analytical models* to characterize both the environment and the workloads[17] and predict some metrics for a parallel system[18]. Finally, the third approach is a combination of *a Simulation system together with a Historical scheme* to estimate the execution time of the parallel applications[19]. The majority of these

Regular Paper
This work was supported by the MEyC under Grant No. TIN 2008-05913.
©2011 Springer Science + Business Media, LLC & Science Press, China

studies are focused on estimating the performance of parallel applications in dedicated environments. Unlike these, we are interested in researching new estimation methods oriented to *non-dedicated clusters*.

In general, non-dedicated clusters are inherently unpredictable due to the non-deterministic behavior of local users. However, there are some specific cases where local load has a deterministic behavior, such as the computer lab of any university, where the local load is scheduled temporarily by means of a timetable. Therefore, some kind of prediction about the behavior of the local workload can be extracted. The prediction in this kind of non-dedicated clusters is the main aim of our work. In this kind of environments, the interaction between local and parallel jobs makes that the estimation process becomes complex.

These reasons led us to research new methods to estimate the performance of parallel workloads in non-dedicated clusters and evaluate their performance in relation to different scheduling policies. With this aim, three different estimation mechanisms, based on the techniques described above, are proposed in this paper: a discrete time *Simulation* tool based on analytical models, a *Historical* system and an integration of both (*Hybrid* scheme). In order to evaluate them, they were implemented in a scheduling system oriented to non-dedicated environments named CISNE (Cooperative & Integral Scheduler for Non-dedicated Environments)[8]. A specific aspect of the CISNE system is that it is able to use any of the estimation methods, either in an *on-line* way (real cluster), which means that a new estimation of the turnaround time is made on every job arrival in a real cluster, or in an *off-line* mode (simulated cluster), which estimates the behavior of the parallel workload over simulated environments. Thus, we can evaluate different workloads and scheduling policies over several simulated cluster environments, varying their heterogeneity, size, local workload, computational resources of nodes and so on.

In a preliminary work[19], we presented the initial results for two prediction proposals (*Simulation* and *Historical*) applied to an on-line mode (real cluster). Our proposals were further extended, tuned and simplified and provided better performance results. Likewise, they were extended to be applied to an off-line mode (simulated cluster). Thus, they have been tested in a wide set of workloads and requirements. In addition, this paper extends that work by evaluating our proposals in two different non-dedicated environments, namely a real and a simulated cluster. In these frameworks, we evaluated the effects of different scheduling policies on the estimation methods. In addition, we analyzed the influence of the local load on the estimation methods in both environments. Likewise, we discussed the cost of the estimation carried out by each approach. Finally, we compared our proposals to representative estimation methods in the literature[20-21].

The outline of this paper is as follows. Section 2 shows some work related to the present study. The underlying scheduling environment, named CISNE, is described in Section 3. Our proposals for the on-line estimation process are explained in Section 4. The integration of these estimation kernels in an off-line simulator is described in Section 5. Next, the experimental results are analyzed in Section 6. Finally, conclusions and the future work are explained in Section 7.

2 Related Work

The easiest way to achieve an estimate of the execution time for a given parallel application is to ask its user. However, several studies[22-24] have shown that user estimations are totally inaccurate.

In order to provide better estimations, some kind of prediction system integrated into the job scheduler is needed. The most widely evaluated alternative relies on historical systems[12-14,16]. This kind of system normally looks for a past state that is similar to the current one. This correlation is defined by a comparison function that determines how similar one state is to another. An extension of those systems was proposed by Lafreniere *et al.* in [11]. They proposed a technique of employing both historical application-run data and a user's knowledge of his application enlarged with a regression model for prediction. Likewise, Feitelson *et al.*[23,25-26] proposed the inclusion of the temporal structure of the parallel workload (p.e., number of jobs and users, inter-arrivals time, parallelism,...) to estimate the future execution of the parallel applications using adaptive backfilling. Finally, it is worth mentioning the work of Yang *et al.*[16], which is the most closely related to our work. They proposed a conservative scheduling policy that uses information about expected future variance in resource capabilities to produce more efficient data mapping decisions. Evaluation of several new one-step-ahead and low-overhead time series prediction strategies based on historical data are performed. However, in contrast to our aim of measuring the turnaround time of parallel applications, their predictions are exclusively aimed at calculating the expected resource capability and the expected variance in that value.

Another approach is the use of a simulation engine to represent scheduling environments. However, these simulators also use a historical system for calculating the execution time of the applications. For instance, in

[21], a simulation of the scheduling system is used in conjunction with historical data to estimate both the waiting and execution time of the parallel applications.

Finally, an analytical model could be used for the characterization of both the workload and the underlying system. In fact, in [17] an entire middleware, named PACE, is presented. This is capable of characterizing the parallel workload, together with the underlying hardware on which the applications are to be run. PACE was used by Jarvis *et al.*, in [18], to improve the scheduling of tasks over homogeneous clusters and provide a basis for the higher-level management of grid system resources. In [27], an extended model to be applied in heterogeneous and real time systems is presented. In a similar line, there are the works by Dinda[28-29] and Wolski[30]. Both authors presented a system for predicting the running time of a compute-bound task on a typical shared, unreserved commodity host. The prediction is computed from linear time series prediction of host loads and takes the form of a confidence interval.

Unlike these previous works, we are interested in researching new estimation methods applicable to non-dedicated environments, which are able to predict the turnaround time for parallel applications, even in the presence of a serial workload of the workstation local user, which is not controlled by the scheduling system. To achieve these aims, we propose and evaluate three different estimation kernels, which fit each of the frameworks described in this related work: a historical system, a simulator system based on analytical models and a system that merges a simulator engine for the scheduling process with a central execution time estimation system based on historical schemes.

3 CISNE: A Scheduling Framework Oriented Towards Non-Dedicated Clusters

In order to implement and evaluate our estimation proposals, we need a scheduling system orientated towards non-dedicated environments. With this goal, in previous work we developed the CISNE (Cooperative & Integral Scheduler for Non-dedicated Environments) system[8] as an integral scheduling environment that merges both time and space sharing subsystems. Fig.1 depicts the general architecture of the CISNE system.

In a non-dedicated cluster, there is an interaction between two different kinds of user. On one hand, there is the local user working on each workstation in the cluster, and on the other, the parallel user who executes the parallel workload on the cluster. Taking both into account, the main objective of CISNE is to manage parallel applications in a non-dedicated environment, ensuring benefits for the parallel applications while preserving the local task responsiveness. To achieve this aim, each job launched in the CISNE system follows the steps shown in Fig.1. When a parallel job is submitted to the CISNE system by a parallel user (Step 1), the job waits in a queue (Step 2) until the Queue Manager decides to dispatch it. This decision is taken according to the computational requirements of each parallel job in the queue, together with the Node State received from each node. The Node State includes the local load and the amount of idle computational resources on each node, which in turn defines the set of available nodes for executing parallel jobs. Once a job is selected from the black *Job Queue* (black Step 3), black CISNE must select the best subset from the available nodes to execute it.

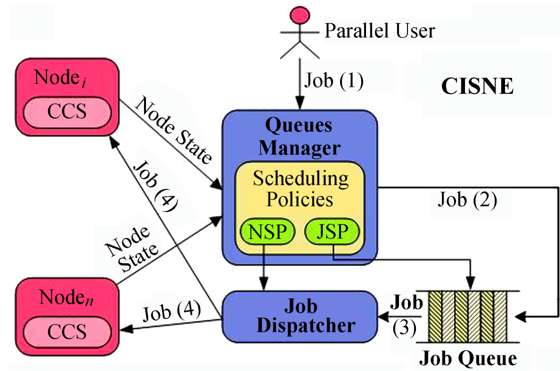

Fig.1. General architecture of the CISNE system.

According to the above description, the CISNE system needs to define the following Job and Node Selection Policies:

• *Job Selection Policy* (JSP) is the policy for selecting the next job to run from the Job Queue. This could depend on the job's priorities (order of the queue), and the estimated cluster state (intrusion level into the local workload, the Multi Programming Level (MPL) of parallel applications throughout the cluster, the Memory and CPU usage on each node and the available nodes). In this work, we have applied the two JSPs most widely used in the literature: one is an FCFS (First-Come-First-Served) alone, and the other is an FCFS augmented with an *Easy Backfilling* technique. An *Easy Backfilling*[31-32] policy consists of executing a job, not at-the-head of the FCFS queue, whenever this does not delay the start of the job at the head.

• *Node Selection Policy* (NSP) is the policy for distributing the parallel tasks among the nodes. This depends on the cluster state and the parallel job characteristics. In this work, we used two different NSP policies, defined in previous works[8], which are designed for non-dedicated environments. The first one, termed Normal, selects the nodes for executing a parallel application considering only the resource usage level

throughout the cluster, so it does not overload any node in detriment of the local user interactiveness. In order to preserve the performance of the local user's applications on each workstation, it establishes an acceptable system usage limit for some computational resources (CPU and Memory) by means of a *social contract*[33] between the local and parallel users, and this is assumed to be equal across the cluster. Nevertheless, using the Normal policy, we are still not considering the parallel and local load interaction inside a node. Therefore, we need to add new characteristics to the scheduling decision process carried out by the Normal policy by defining a new policy, termed *Uniform*. This second policy is characterized by the following restrictions: (a) it executes tasks from different resource-bound applications (i.e., communication or computation bound) in the same node and (b) the set of nodes assigned to different kinds of application should be as equal as possible. Fig.2(a) shows how the Uniform policy executes a CPU bound application (J_3) in the same set of nodes as a communication bound application (J_2). In contrast, in Fig.2(b) a Normal policy executes the J_3 application regardless of the load and its resource-bound property. In this case, the computational resources assigned to J_3 are not the same for all of its tasks. However, in both cases, the computation resources used by the parallel applications do not exceed the limit fixed by the social contract.

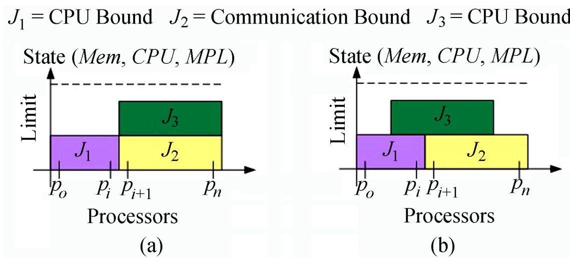

Fig.2. Scheduling difference between the Uniform and Normal policies. (a) Uniform. (b) Normal.

Likewise, and considering that different kinds of parallel applications could reach the system (e.g., PVM or MPI), a specific module, named *Job Dispatcher*, is defined to execute the parallel applications, according to their nature, to the cluster. Once the job is executed (Step 4 of Fig.1), the time sharing modules of CISNE residing on each node, named CCS (Cooperating CoScheduling), take control of the progression of each parallel job. CCS is based on the implicit coscheduling technique, which identifies the processes in need of coscheduling during execution by gathering and analyzing implicit run-time information. In addition to traditional implicit techniques, CCS provides a social contract based on reserving a percentage of CPU and memory resources to ensure the progress of parallel jobs without disturbing the local users, while coscheduling of communicating tasks is ensured. A detailed description of the CCS policy can be found in [34].

4 On-Line Estimation in the CISNE System

In this section, we explain three different estimation kernels (Historical, Simulation and Hybrid) to predict on runtime the performance of a given parallel workload in a non-dedicated cluster system. In addition, we describe how each of these methods is integrated into the CISNE system.

4.1 Estimation Through Historical Data

Our first approach is based on a historical system that records past executions and tries to infer the future as a replication of the past. The integration of such a system into CISNE is depicted in Fig.3. As we can see in this figure, the historical system is made up of a historical repository (named *HistDB*), which registers some information about the system state (jobs, queue and cluster), and an estimation system (named *Estimator*), which deals with the prediction of the turnaround time from the information stored in HistDB. Note that a new prediction is triggered whenever a new job is submitted to the CISNE system (Step 2 of Fig.3).

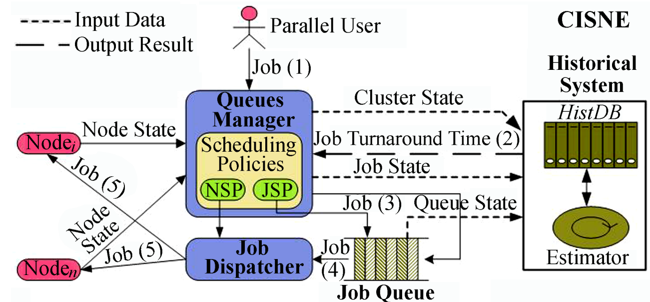

Fig.3. Integration of the historical method into the CISNE system.

The information, collected by CISNE on runtime, includes the following:

• *Job State*. The historical system records some information about each executed job, such as its arrival, waiting and execution time; the name of the application and its input parameters, the configured JSP and NSP scheduling policies being used by CISNE and the amount of CPU time used by this job.

• *Queue State*. For every submitted job, the system records information about the jobs waiting in the jobs queue at two different moments: when the job is submitted and when the job is selected for execution according to the JSP policy.

• *Cluster State*. This is also recorded whenever a new job starts its execution. It includes a list of nodes where the job is executed, together with the amount of CPU and memory resources used by the local and parallel loads running in such nodes.

It is worth pointing out that the need to estimate a system as complex as a non-dedicated cluster implies that CISNE has to collect more detailed information than the other historical systems described in the literature[11,13,20-21], which are oriented towards dedicated systems. Table 1 compares the information gathered by these authors in relation to our proposal. For example, we can see as Lafreniere[11] only stores information about previously executed jobs jointly with their input parameters and the number of nodes used by the executed applications. Likewise, Gibbons[13] uses a set of templates to separate different stored cases of executed jobs. These templates are defined using the queue where the jobs were submitted. This is very similar to the *Queue state* registered by CISNE on each new job submission.

Table 1. Relation of Gathered Information for Different Authors Compared with This Work

Author	Information
Gibbons[8]	[u, e, n, rtime] [u, e] [n, rtime] [e] [n, rtime]
Li[20]	[g, u, q, e, n]
Smith[30]	[u, e, n] [q, u, n]
Lafreniere[19]	[u, n, ip]
This work	[q, e, n, ip, rtime]

Note: g: group, u: user, q: queue, e: executable, n: number of nodes, ip: input parameters, rtime: running time, [ ]: templates.

The second point to emphasize is how CISNE manages the stored information. In this sense, it is worth pointing out that this process should spend a magnitude of time much lower than the metric being estimated. Likewise, the estimation error should be lower than an acceptable threshold. Taking into account the studies of [14, 20], such an acceptable threshold was fixed at 40%. Experimentally, we tested that both parameters (data processing time and estimation error) are balanced below these desirable thresholds, whenever the HistDB is populated with 10 000 records and each search uses around 50 records to retrieve the estimation information. Note that these records were collected over a 6-month period, using data from the execution of the set of applications described in Section 6 of this paper.

The next subsection describes the estimation process of the turnaround time followed by CISNE's Estimator.

Turnaround Time Prediction

In order to estimate the turnaround time for a given parallel job, the Estimator needs to predict two different metrics separately. On one hand, there is the time that the job will spend in the jobs queue (*waiting time*) and, on the other hand, the time that the job will spend during its future execution (*execution time*).

Prior to this, there are some concepts that deserve explanation:

• *Application* (denoted as *Appl*) is the program being executed by a job. It is defined by the executable, its input parameters and the number of processors needed.

• *Job* (denoted as J) is a particular execution of a given *Appl*. It depends on the scheduling policy, the cluster state and the other jobs executed concurrently with the job.

• $getCPUtime(HistDB, Appl, Policy)$ is a function that returns from the *HistDB* the average CPU time used by an *Appl* for a given scheduling policy (*Policy*). We assume that the CPU time remains constant for a given application.

• $Waiting_Jobs_{\mathrm{ahead}}(J)$ is the whole set of jobs in the job queue, when job J is submitted to the queue.

• $Waiting_Jobs_{\mathrm{backwards}}(J)$ is the whole set of jobs in the jobs queue, when the job J is executed by the system.

• $CPUtime_{\mathrm{ahead}}(J)$ is the sum of the *CPUtime* associated with each job belonging to the $Waiting_Jobs_{\mathrm{ahead}}(J)$ set.

• $getCPUtime_{\mathrm{ahead}}(HistDB, State, J)$ is a function that retrieves the $CPUtime_{\mathrm{ahead}}$ of every job J stored in *HistDB*, considering the current *Waiting* (i.e., the number and type of jobs currently waiting in the queue).

• $CPUtime_{\mathrm{backwards}}(J)$ is the sum of the *CPUtime* associated with each job belonging to the $Waiting_Jobs_{\mathrm{backwards}}(J)$ set.

Note that both $CPUtime_{\mathrm{ahead/backwards}}(J)$ metrics are used as search patterns across the historical database.

Algorithm 1. Historical Estimation Process for the Waiting Time of *CurrentJob*

1: **forall** (J in $Waiting_Jobs_{\mathrm{ahead}}(CurrentJob)$)
2: $Appl = getAppl(J)$
3: $CPUtime_{\mathrm{ahead}} = CPUtime_{\mathrm{ahead}} + getCPUtime(HistDB, Appl, Policy)$
4: **end forall**
5: $SimilarJob = J \in HistDB$ such that $getAppl(J) == getAppl(CurrentJob)$ and minimizes $|CPUtime_{\mathrm{ahead}} - getCPUtime_{\mathrm{ahead}}(HistDB, State, J)|$
6: **return** $WaitTime = getWaitTime(HistDB, SimilarJob)$

Algorithm 1 describes the method used to estimate the *waiting time* for a new job submitted to the

system (*CurrentJob*) by means of the historical repository (*HistDB*). First, Algorithm 1, lines 1∼4 calculates the $CPUtime_{\rm ahead}$ associated with the *CurrentJob*. Note that this calculation implies that the Estimator looks into the *HistDB* for the *CPUtime* of the application *Appl* associated with each job J belonging to the jobs queue. Next, Algorithm 1, line 5 looks into the *HistDB* for the job (*SimilarJob*) with the most similar $CPUtime_{\rm ahead}$ to the calculated one. This search is done among those jobs which are associated with the same application ($getAppl(J)$) as *CurrentJob*. Thus, the search time is considerably reduced. Note that it also takes the scheduling policy and local load (*state*) into account. Finally, the waiting time for a *SimilarJob* is returned as the estimated waiting time for the *CurrentJob*.

It should be noticed that the complexity of Algorithm 1 depends linearly on the number of jobs in the job queue (N_{Job_Queue}), and the number of records in the *Hist*DB (N_{HistDB}). Given that N_{HistDB} is several orders of magnitude greater than N_{Job_Queue}, the resulting order is linear with N_{HistDB} ($O(N_{HistDB})$).

The second step in the *turnaround* estimation is the prediction of the *execution time* of *CurrentJob*. Algorithm 2 depicts the steps for such an estimate. It is worth pointing out that we have to estimate the execution time of *CurrentJob* without knowing the state of the environment when it was finally executed. Hence, the first step is to estimate such an environment (Algorithm 2, lines 1∼5). Following the same reasoning used in Algorithm 1, this estimation is done from the $CPUtime_{\rm ahead}$ metric. From the jobs stored in *HistDB*, we build a list of jobs (*SimilarJobList*), whose $CPUtime_{\rm ahead}$ is similar to the *CurrentJob*. Notice that for our studies we will use a *threshold* = 5% given that this value was found to be a good compromise between the time needed to process enough data for doing the estimation and the amount of data retrieved from the *HistDB*. After that, we try to estimate the amount of CPU time that the *CurrentJob* will need to compete with when it is finally executed. With this aim, the $CPUtime_{\rm backwards}$ associated with each job belonging to *SimilarJobList* is averaged in Algorithm 2, line 6. Next, we look among the *SimilarJobList* set for the job (*SimilarJob*) whose $CPUtime_{\rm backwards}$ is the closest (Algorithm 2, line 7) to the average calculated in Algorithm 2, line 6. The execution time for *SimilarJob* is then returned as the estimated execution time of *CurrentJob*.

It is noticed that the relevance of calculating $CPUtime_{\rm backwards}$ is given by the fact that we are dealing with a non-dedicated environment. This implies that the same node is shared among several applications, and hence the execution time of each one affect the others. Thus, an application that is going to be started later could affect the execution time of an application started sooner.

Algorithm 2. Historical Estimation Process for the Execution Time of *CurrentJob*

1: **forall** (J in $Waiting_Jobs_{\rm ahead}(CurrentJob)$)
2: $Appl = getAppl(J)$
3: $CPUtime_{\rm ahead} = CPUtime_{\rm ahead} + getCPUtime$ $(HistDB, Appl, Policy)$
4: **end forall**
5: $SimilarJobList = \{every\ J \in HistDB$ such that $getAppl(J) == getAppl\ (CurrentJob)$ and $|CPUtime_{\rm ahead} - getCPUtime_{\rm ahead}(HistDB,\ State,\ J)| < threshold\}$
6: **forall** (J in *SimilarJobList*)
$AverageCPUtime =$
$\mathrm{Average}(getCPUtime_{\rm backwards}(HistDB,\ State,\ J))$
7: $SimilarJob = J \in SimilarJobList$ such that minimizes $|AverageCPUtime - getCPUtime_{\rm backwards}$ $(HistDB, State, J)|$
8: **return** $ExecTime = getExecTime(HistDB,\ SimilarJob)$

It is worth pointing out that the complexity of Algorithm 2 depends on the number of elements in the *HistDB* again ($O(N_{HistDB})$). In this sense, note that the estimation process of the waiting (Algorithm 1) and execution time (Algorithm 2) are related, because both algorithms use the $CPUtime_{\rm ahead}$ metric for the estimation process. In fact, the results for the waiting time are reused for the execution time calculation in the real implementation. This causes a considerable reduction in the calculation time of the estimation process. Likewise, the fact that our system collects more complete historical information than other systems[11,13,21] based on historical repository allows us to use a simpler estimation process than previous works.

4.2 Estimation Through Simulation

The second alternative is a discrete time simulation schema based on an analytical model as its core turnaround time prediction engine. In order to deal with the estimation in a non-dedicated environment, the simulator needs two different kinds of information: the characterization of the parallel applications and the modeling of the current cluster state, including the local load activity. As we can see in Fig.4, this information is provided by the *Application Characterization* and the *Queue Manager* modules of CISNE, respectively. At the end of the simulation, the estimated job turnaround time is returned to the Queue Manager. This information is then used by the scheduler and returned to the parallel user.

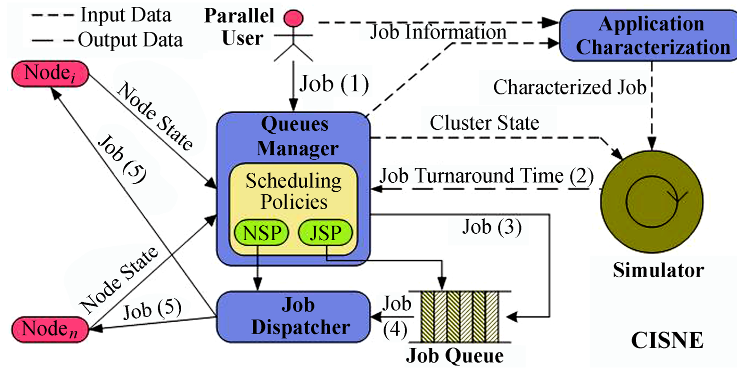

Fig.4. Integration of the simulation process into the CISNE system.

The Application Characterization module of CISNE can obtain the behavior of the parallel applications in two different ways. The first option, which is the most widely used, is to run the parallel application in isolation, using a particular scheduling policy and a specific set of nodes where the application can run alone without the influence of other jobs. The resulting resource consumption metrics are then stored by the Application Characterization module to be used for future estimations. Whenever it is not possible to run a parallel application in isolation, a second choice is to ask the parallel user about the resources used by their application. However, giving that the information given by the parallel user is normally an inaccurate method[22], CISNE applies a readjustment based on the historical information collected on each new real execution. Basically, this readjustment is based on the estimation process explained in Subsection 4.1 but assuming a dedicated environment for the parallel application.

For whatever option and for a fixed number of processors per job (n), the Application Characterization module collects:

• $ExecTime_{\rm total}(J)$ is the execution time of job J;

• $CPUtime_{\rm total}(J)$ is the amount of CPU time used by job J;

• $CPU(J)$ is the CPU percentage ($CPUtime_{\rm total}(J)/ ExecTime_{\rm total}(J)$) used by job J.

Whenever a new job is submitted to the CISNE system, the Queue Manager collects the usage of the resources in each node, together with the state of each job J running in the cluster. The following set of data models the cluster state:

• *JSP and NSP policies* are the Job and Node Selection Policies used by CISNE, respectively;

• $ExecTime_{\rm cur}(J)$ is the current running time for job J;

• $CPUtime_{\rm cur}(J)$ is the amount of CPU time used by the job J from its beginning;

• $nodes(J)$ is the set of nodes where job J is running;

• $CPU_{\rm local/paral}(n)$ is the sum of the CPU required percentage of each local/parallel task running in the node n; (If this value is higher than 1 it means that the node n is overloaded.)

• $MPL_{\rm local/paral}(n)$ is the number of local/parallel tasks executing simultaneously in the node n. As we demonstrated in several previous studies[8,34], the time sharing component of CISNE allows the execution of more than one parallel job in the same set of nodes, whenever it does not disturb the local user. Note that the maximum $MPL_{\rm paral}(n)$ through the cluster will be denoted as $MPL_{\rm paral}$.

• $tasks(n)$ is the set of parallel tasks running in the node n.

Once all the needed elements have been collected, CISNE is ready to start the simulation process described in the next subsection.

4.2.1 Simulation Algorithm

The discrete time simulation process is triggered whenever a new job reaches the CISNE system. Every time that the simulation is started, the turnaround time for each job in the system, either running or waiting, is estimated. This process provides extra information about the future cluster state to the scheduler (Step 2 in Fig.4). Using the estimated cluster state, it is possible to enhance the *NSP* and *JSP* scheduling decisions. On one hand, the *NSP* policies use this estimation to avoid surpassing the resource utilization limit defined by the social contract. On the other hand, the *Backfilling JSP* policy could take benefit from the cluster state estimation to find new scheduling opportunities by advancing the execution of some waiting jobs. Notice that if the simulator is working when a new job arrives, the whole process has to be restarted to consider the new job to be executed. Algorithm 3 depicts our simulation method.

The core of the simulation algorithm relies on a *while* that loops as long as any parallel job is running (Algorithm 3, lines 3∼16). For each loop, the algorithm

estimates the *Remaining Execution Time* (*RemTime*) of every job in the running queue (DRQ), selects the next job that will finish (J_i) and removes it from DRQ (Algorithm 3, lines 4∼6). After that, the $CPUtime_{\rm cur}$ used by each of the remaining jobs in DRQ is calculated (Algorithm 3, line 7) to be used in the following simulation step (Algorithm 3, line 3 loop). Next, another nested loop tries to execute some waiting jobs using the system scheduling policy, the available resources in t_i and those resources released by J_i (Algorithm 3, lines 8∼13). Finally, the waiting time for every job in the Waiting Queue (DWQ) is updated (Algorithm 3, line 14) and the simulation step advances to t_i (Algorithm 3, line 15) in the discrete time process.

Algorithm 3. Simulation Process

1: Duplicate the system state in a *dummy* system state: Dummy Waiting Queue (DWQ) as a copy of the jobs waiting queue, Dummy Ready Queue (DRQ) as a copy of the running jobs queue and $Cl_{\rm sim}$ as a copy of the cluster nodes with their state.
2: Store the current time (t_0), as the simulation start-time.
3: **while** ($\exists J$ in DRQ) **do**
4: **forall** (J in DRQ) Calculate the $RemTime(J)$.
5: Assume that the job J_i is the next one to finish in time t_i.
6: Update the estimated $ExecTime_{\rm total}(J_i)$ to t_i and remove J_i from DRQ.
7: **forall** (J in DRQ) Calculate the $CPUtime_{\rm cur}(J)$ in $[t_0, t_i]$.
8: **while** ($\exists$ usable resources in $Cl_{\rm sim}$ and any job waiting in DWQ) **do**
9: Look for an job J_x in DWQ that could be executed in the $Cl_{\rm sim}$ state.
10: Select the best subset of $Cl_{\rm sim}$ for executing J_x, using the system policy.
11: Execute the job J_x in the selected subset of $Cl_{\rm sim}$ and add it to DRQ.
12: Increment the estimated $WaitTime(J_x)$ in $[t_0, t_i]$.
13: **endwhile**
14: **forall** (J in DWQ) Increment the estimated $WaitTime(J)$ in $[t_0, t_i]$.
15: Set t_0 to t_i.
16: **end while**

Regarding the complexity of the simulation algorithm, we can see that there is a loop nesting (Algorithm 3, lines 4, 7, 8 and 14 inside main loop: line 3). The order of the main loop is linear with the number of running jobs (N_{DRQ}), whereas the order of the longest nested loop (Algorithm 3, line 8) depends on the waiting jobs (N_{DWQ}) and the cluster size (S_{cl}). Taking into account the characteristics of a non-dedicated system, we can assume that the size of both sets, DRQ and DWQ, are of the same magnitude[35] and as a consequence, the resulting complexity is quadratic with the number of jobs in the running queue and the size of the cluster ($O(N_{DRQ}^2 \times S_{cl})$).

In order to carry out this simulation, we need a pair of extra functions which define the estimation process. The first is the *RemTime* (Algorithm 3, line 4), which estimates the *Remaining Execution Time* for a given job considering the current cluster and job state. The second tries to predict the *CPU time* ($CPUtime_{\rm cur}$) that the job has used from the beginning until the moment when the simulation is launched (Algorithm 3, line 7). Our approaches to solving both functions are depicted in the following subsections.

4.2.2 Remaining Execution Time Approaches

Our first approach, named Proportional (*Prop*), is based on the belief that the future will be similar to the past. Therefore, the remaining execution time of a job J, denoted as $RemTime(J)$, is calculated according to the following (1) and (2):

$$CPUtime_{\rm rem}(J) = CPUtime_{\rm total}(J) - CPUtime_{\rm cur}(J), \quad (1)$$

$$RemTime(J) = \frac{ExecTime_{\rm cur}(J) \times CPUtime_{\rm rem}(J)}{CPUtime_{\rm cur}(J)}. \quad (2)$$

Note that (1) and (2) assume that the CPU time ($CPUtime_{\rm total}(J)$) used by job J during its complete execution ($RemTime(J) + ExecTime_{\rm cur}(J)$) is proportional to the CPU time ($CPUtime_{\rm cur}(J)$) used during the current execution time ($ExecTime_{\rm cur}(J)$).

The second proposal, denoted as *MPL*, considers both the past and current states. It starts by calculating the remaining execution time that job J would need if it was executed in isolation ($RemTime_{\rm isol}(J)$). This value, following the same reasoning as (2), is calculated as:

$$RemTime_{\rm isol}(J) = \frac{ExecTime_{\rm total}(J) \times CPUtime_{\rm rem}(J)}{CPUtime_{\rm total}(J)}, \quad (3)$$

where $CPUtime_{\rm rem}$ is calculated according to (1). Likewise, we calculated the maximum *MPL* ($MPL_{\rm max}(J)$) as follows:

$$MPL_{\rm max}(J) = \max_{\forall n \in nodes(J)} (MPL_{\rm paral}(n) + MPL_{\rm local}(n)). \quad (4)$$

It is worth pointing out that (4) returns the maximum number of tasks, both local ($MPL_{\text{local}}(n)$) and parallel ($MPL_{\text{paral}}(n)$), executing concurrently with job J among the nodes where it is running ($nodes(J)$). Taking $MPL_{\max}(J)$ and $RemTime_{\text{isol}}(J)$ into account, the remaining execution time for J is calculated according to the following equation:

$$RemTime(J) = RemTime_{\text{isol}}(J) \times MPL_{\max}(J). \quad (5)$$

In this way, the *RemTime* is proportional to the current *MPL* throughout the cluster.

Our last approach, denoted as *CPU*, considers not only the number of tasks executing concurrently (*MPL*) but also the *CPU requirements* of those tasks (in percentage). So, the $RemTime(J)$ is calculated as follows:

$$RemTime(J) = RemTime_{\text{isol}}(J) \times \frac{CPU(J)}{CPU_{\text{feas}}(J)}, \quad (6)$$

where:

$$CPU_{\text{feas}}(J) = \min(CPU(J), \frac{CPU(J)}{CPU_{\max}(J)}), \quad (7)$$

is the *feasible* CPU percentage that job J could use. Notice that $CPU_{\text{feas}}(J)$ could be at most the amount of CPU required by job J, but this percentage could be decreased if some of the nodes shared by J has some other load. This load is defined by $CPU_{\max}(J)$ as:

$$CPU_{\max}(J) = \max_{\forall n \in nodes(J)} (CPU_{\text{paral}}(n) + CPU_{\text{local}}(n)). \quad (8)$$

Thus, $CPU_{\max}(J)$ is the maximum CPU usage requirements (in percentage) among the nodes (n) where J is running ($nodes(J)$).

It is important to emphasize that no matter which the chosen approach is, the value for $CPUtime_{\text{cur}}(J)$ is only real at the beginning of the simulation process (Algorithm 3, line 3), and for each simulation step it is necessary to estimate this value (Algorithm 3, line 7) again. Therefore, in the next subsection, we describe some proposals for estimating this value.

4.2.3 Current-CPU Time Proposals

This subsection describes two different proposals for estimating the $CPUtime_{\text{cur}}$ for a given job J at a specific moment (t_i), denoted as $CPUtime_{\text{cur}}(J, t_i)$, considering that this value has been measured or estimated in the past ($CPUtime_{\text{cur}}(J, t_{i-1})$).

In our first approach, denoted as *MPL*, we assume that the job CPU usage is proportional to the maximum *MPL* calculated in (4). The following expressions represent this proposal.

$$\triangle\, CPUtime = \frac{(t_i - t_{i-1}) \times CPUtime_{\text{total}}(J)}{MPL_{\max}(J) \times ExecTime_{\text{total}}(J)}, \quad (9)$$

$$CPUtime_{\text{cur}}(J, t_i) = CPUtime_{\text{cur}}(J, t_{i-1}) + \triangle CPUtime. \quad (10)$$

Our second proposal, denoted as *CPU*, is based on the same idea used for the Remaining Time in (6), but applied to the *CPUtime*. The following equations represent that idea.

$$\triangle\, CPU_{\text{feas}} = (t_i - t_{i-1}) \times CPU_{\text{feas}}(J), \quad (11)$$

$$CPUtime_{\text{cur}}(J, t_i) = CPUtime_{\text{cur}}(J, t_{i-1}) + \triangle CPU_{\text{feas}}. \quad (12)$$

4.3 Estimation Through a Hybrid System

Our last proposal combines both alternatives presented above (Historic and Simulation systems). This integration can be seen in Fig.5, where the simulation engine uses historical information to make its estimates. It is worth pointing out that the Application Characterization module (Apps. Charact. in the figure) is integrated into the Historical system. This integration

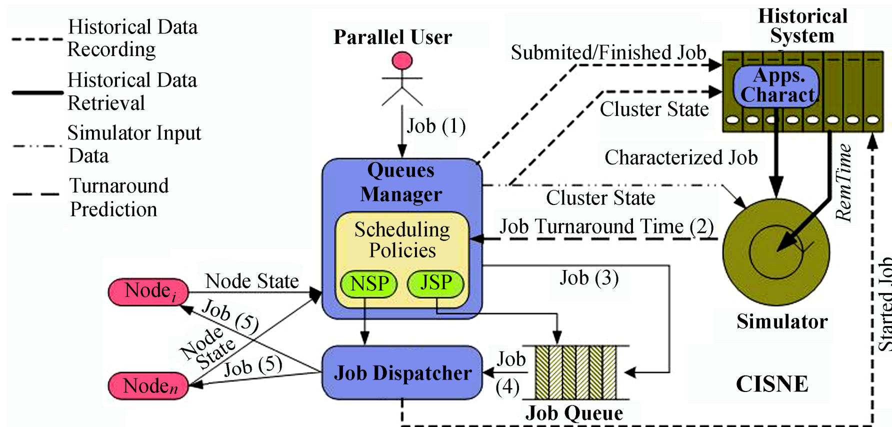

Fig.5. Integration of the hybrid schema into the CISNE system.

endows the module with a kind of memory, which allows the parallel jobs to be characterized much better.

The main advantage of merging the Simulator and the Historical systems resides in the way the *Remaining Execution Time* (*RemTime*) of each job is obtained. In this case, it is calculated through a historical information retrieval system instead of the analytical model described in Subsection 4.4.2. The proposed *RemTime* method, described in Algorithm 4, is based on obtaining the number of tasks that share resources (*SharedTasksCount*) with the job being estimated (*CurrentJob*). This metric is used as a search pattern for historical matching. Thus, we can find a historical scenario, where *CurrentJob* was run with a similar load in its assigned nodes as in the present time, from which we can expect a similar running time. According to this aim, the algorithm looks for the set of executions (jobs) in the historical repository (*HistDB*) associated with the same application as *CurrentJob*, which have the same *SharedTasksCount* (Algorithm 4 line 8). From this set of jobs, Algorithm 4 line 9 averages their execution time. Next, this average is subtracted from the actual execution time for *CurrentJob* ($ExecTime_{\mathrm{cur}}$, Algorithm 4, line 13), and returned to the simulation engine as the remaining time (*RemTime*) of the *CurrentJob*. Note that if the subtraction is a negative value (Algorithm 4, line 10) then 0 is returned to the simulator indicating that the job should have finished.

Algorithm 4. Historical Estimation Process for the Remaining Execution Time in the Hybrid System

1: *SharedTasksCount* = the number of tasks that share some node with *CurrentJob*.
2: *CurrentAppl* = *getAppl*(*CurrentJob*)
3: **if** *SharedTasksCount* == 0 **then**
4:     **forall** ($J \in$ *HistDB* such that *CurrentAppl* == *getAppl*(J) and it was run in isolation)
5:         *ExecTime* = Average(*getExecTime*(*HistDB*, *State*, J))
6:     **return** $ExecTime - ExecTime_{\mathrm{cur}}(CurrentJob)$
7: **else**
8:     *SimilarJobList* = {every $J \in$ *HistDB* such that *CurrentAppl* == *getAppl*(J) and *SharedTasksCount* == *getSharedTasksCount*(*HistDB*, *State*, J)}
9: **forall** (J in *SimilarJobList*) *ExecTime* = Average(*getExecTime*(*HistDB*, *State*, J))
10:     **if** $ExecTime - ExecTime_{\mathrm{cur}}(CurrentJob) < 0$ **then**
11:         **return** 0
12:     **else**
13:         **return** $ExecTime - ExecTime_{\mathrm{cur}}(CurrentJob)$
14: **end if**

In relation to the complexity of the Hybrid system, we should take into account that we have to replace the analytical method used by the Simulation system (see Algorithm 3, line 4) for the searching method in the historical repository explained in this subsection (see Algorithm 4). It means that the quadratic complexity of the Simulation system is increased by the linearity associated with the Historical estimator. Hence, the resulting order of the algorithm is $O(N_{DRQ}^2 \times S_{cl} \times N_{HistDB})$, where N_{DRQ} is the number of jobs running in the system, S_{cl} is the cluster size and N_{HistDB} is the number of records in the *HistDB*.

5 Off-Line Estimation in the CISNE System

The on-line estimation, described in Section 4, allows the performance of a parallel workload for a given cluster environment to be known. However, there are some situations where the parallel user or the system administrator are interested in evaluating the performance of a given parallel workload in several cluster environments. It means having available the usefulness of simulating, not only the behavior of a certain parallel workload, but also several cluster environments, vary their heterogeneity, size, local load, computational resources of nodes and so on. With this in mind, an *off-line simulation* system has been included in the CISNE system.

With this aim, the Job Dispatcher together with the nodes that integrate the CISNE system have been replaced by a set of modules that simulate the functionality of the original ones giving the possibility of emulating resources that are not really available. Thus, any scheduling policy is perfectly usable in both real or simulated environments. In Fig.6, we can observe the modules that are replaced by others, which are named dummies. They are provided to the Queue Manager with an environment where it can schedule jobs as if these are executed on a real cluster. Note that all the dummy modules are under the control of the off-line simulator, by means of an Event Queue that stores the events that should appear in the future, such as job (both local and parallel) arrivals, execution and finalization.

In order to initially fill up the Event Queue, the Extern Event Generator takes, as an input, a set of local and parallel applications, which are described in the *Parallel/Local Load Configuration file* (see Fig.6). For each parallel/local job, its corresponding arrival time and expected execution time in a dedicated cluster are provided. It is important to remark that the job description provided to the *Extern Event Generator* are

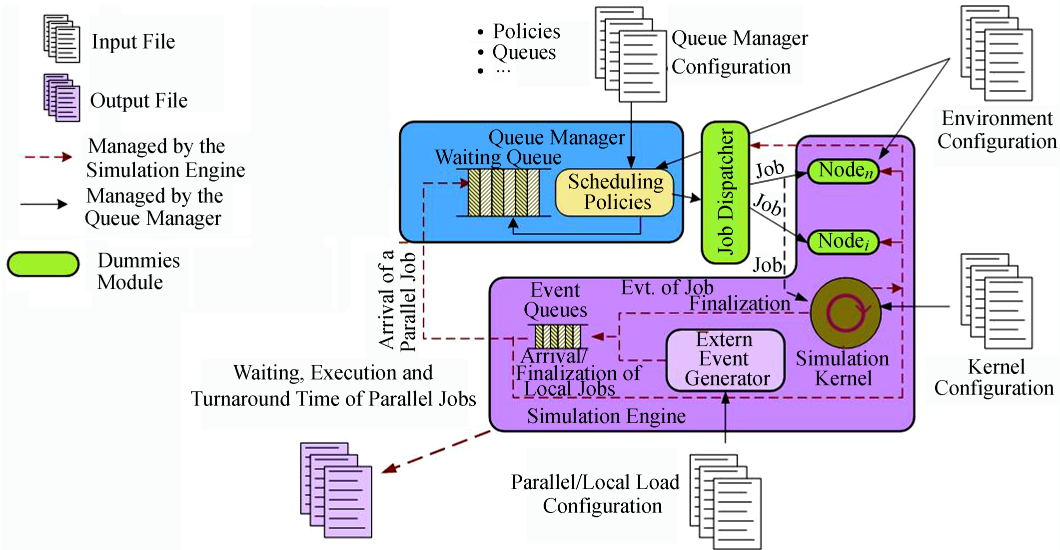

Fig.6. Architecture of the off-line simulator.

the same used for the *on-line* estimation (see Subsection 4.2).

Once the Event Queue is loaded with the arrivals of the local and parallel jobs, the off-line simulator starts by taking the jobs arrival events in order. Following the established scheduling policy, the jobs are distributed among the dummy nodes for execution or pushed into the waiting queue if there are not enough free resources for their execution. Every time that a new job is started in a set of nodes, specified by the *Environment Configuration file*, the Simulation Kernel estimates its execution time. Notice, that the *Environment Configuration* defines node parameters such as: total memory or initial load when the scheduling starts, regarding CPU and memory usage. Likewise, the Simulation Kernel reestimates the execution time of any job that shares at least one node with the just launched node. Using these estimations, a new finishing event is added to the Event Queue (for the launched job), while some others may be modified (for the jobs sharing some nodes with the launched job). Whenever a finishing job event is processed by the off-line simulator, a set of resources are freed and hence the scheduling process is restarted using the new cluster state. Thus, a new set of waiting jobs can be executed and the whole simulation cycle is restarted again. This process is repeated until the Events Queue is empty. Finally, the Simulation Engine returns the waiting, execution and turnaround time of each simulated parallel job to an output file together with the makespan associated with the whole workload.

This way of operation makes the off-line simulator a *discrete event simulation* environment, that includes any of the turnaround time predictors (*kernels*) based on simulation (*Simulation* based on Analytical Models *or Hybrid* ones) defined in Subsections 4.2 and 4.3, respectively. In this way, the same estimation kernels are reused for the on-line prediction as they are for the off-line simulation. Therefore, the results obtained by means of the off-line simulation are directly compared to the ones achieved by the estimation carried out in a production environment.

6 Experimentation

The performance of our prediction proposals was evaluated in two different environments, a real and a simulated non-dedicated cluster. The real environment allowed the accuracy of the on-line estimation methods (Simulation, Historical and Hybrid) to be evaluated in relation to different scheduling strategies, and vice-versa. Likewise, we compared our proposals to other estimation methods described in the literature. Finally, we simulated the same real environment to measure the good behavior of the off-line simulator.

6.1 Real Environment

This subsection discusses the performance of our estimation proposals in a real controlled cluster. With this aim, the parallel and serial workloads and the set of scheduling policies used in our experimentation are described. Next, the advantages and drawbacks of the proposed methods are explained in relation to the results achieved. Finally, an analysis of the time cost/complexity of these estimation methods is included.

6.1.1 Workload and Policies

In order to carry out the experimentation process,

we need two different kinds of workloads: local and parallel.

The serial local workload of the workstation owner was carried out by running one synthetic benchmark, called *local*. This benchmark alternates CPU activity with interactivity by running several system calls and different data transfers to memory. This is configured according to three different input parameters: CPU load, memory requirements and network traffic. Thus, we can simulate any kind of local user profile. In order to assign these values in a realistic way, we monitored the average resources used by students doing their practices in a computer lab of our university during a month. According to this monitoring, we defined three local user profiles, characterized in Table 2: (*Shell*) a user with high CPU requirements, (*Xwin*) high interactivity requirements and (*Internet*) high communication requirements. Taking this monitoring into account, 75% of the nodes in the cluster used throughout this experimentation were loaded with this local benchmark. The laboratory was rarely completely full. The percentage of nodes with each profile follows the distribution shown in the *Distribution* column of Table 2. In order to simulate the variability of local user behavior, the local execution time was modeled by a two-stage hyper-exponential distribution with means, by default, of 60 and 120 minutes and a weight of 0.4 and 0.6 for each stage[36]. These values are due to the fact that the daily timetable of these labs is divided into slots of 60 or 120 minutes. Thus, the behavior of the local users during these time slots is fairly predictable. Note that each new generated *local* benchmark was launched in a node without any local users.

The parallel workload was a list of 1000 PVM/MPI NAS[37] parallel jobs (CG, IS, MG and BT) with a size of 4, 8 or 16 tasks. Table 3 shows the CPU percentage, memory size, communication rate and execution time for 4, 8 and 16 nodes for each NAS benchmark. The jobs making up the parallel workload together with their size were chosen according to a uniform distribution. According to the parallel workload model of Lublin *et al.*[38], each chosen job reached the system following a gamma distribution with $\alpha = 12$ and $\beta = 2.55$. Note that these parameters model the arrival process during the daily cycle of a SuperComputing Center (Institute of Technology in Stockholm). It is important to mention that the maximum number of parallel tasks per node (MultiProgramming Level, $MPL_{\rm paral}$) reached for the workload through the cluster depends on the system state at each moment, but in no case will surpass an $MPL_{\rm paral} = 4$. This is established in order to respect the social contract with the local user[34]. It is worth pointing out that with these parameters we achieved a mean length of the waiting queue of 4 parallel jobs.

This workload was executed with different combinations of Job Selection (*JSP*) and Node Selection policies (*NSP*). Specifically, we combined both JSP policies, FCFS and Backfilling (BF), with both NSP policies, Normal and Uniform. All of them are explained in detail in Section 3. Finally, for the purpose of comparison, we included a *Basic* policy made up of a *Normal*+ *FCFS* policy with an $MPL_{\rm paral} = 1$ (the $MPL_{\rm paral}$ was restricted to 4 in the rest of the evaluated policies).

The whole system was evaluated in an Linux cluster using 64 P-IV (1.8 GHz) nodes with 1GB of memory and a fast Ethernet interconnection network. In order to measure the accuracy of our prediction, we used the normalized average of the absolute prediction errors (D_v) of a given *metric* (turnaround, execution or waiting time). So, $D_v(metric)$ will be calculated as follows:

$$D_v(metric)\% = \frac{\sum_{i=1}^{1000} |Y_i - X_i|}{\sum_{i=1}^{1000} X_i} \times 100\%, \qquad (13)$$

where Y_i and X_i are the predicted and the observed

Table 2. Local User Requirements (The standard deviation is shown between brackets.)

Local	Distribution (%)	CPU_Load	Memory (%)	Network (Bytes/s) Rec.-Send
Shell	15	0.40 (0.2)	20 (15)	108-3 (30-1)
Xwin	62	0.15 (0.1)	35 (55)	608-30 (302-5)
Internet	23	0.20 (0.1)	60 (75)	3154-496 (1890-245)

Table 3. Characterization of the Parallel Workload

Benchmarks	CPU (%) 4/8/16	Memory (MB) 4/8/16	Network Rate (%) 4/8/16	Exec. Time (s) 4/8/16
IS	58/25/24	380/260/150	42/75/76	280/240/179
MG	90/78/70	220/113/60	10/22/30	209/119/75
CG	72/61/52	112/112/112	28/39/48	465/395/375
BT	96/92/87	22/14/8	4/8/13	775/512/246

values, respectively, for each executed application and for a given *metric* (turnaround, execution or waiting time).

6.1.2 Experimental Results in a Real Cluster

First, the performance of the different analytical proposals for calculating the *RemTime* (*PROP*, *MPL* and *CPU*) and the $CPUtime_{\mathrm{cur}}$ (*MPL* and *CPU*), explained in Subsection 4.2, together with the Historical and the Hybrid methods, described in Subsections 4.1 and 4.3 respectively, were evaluated under the simplest scheduling policy defined in this experimentation, namely the Basic policy. Fig.7 shows the error on estimating the turnaround time of the parallel jobs from applying the Basic restrictions (FCFS of Job Selection Policy and Normal of Node Selection Policy with $MPL_{\mathrm{paral}} = 1$). According to the explanation of Subsection 6.1.1, 75% of the nodes in the cluster used throughout this experimentation were loaded with local load. The percentage of nodes with each local profile follows the distribution shown in the *Distribution* column of Table 2.

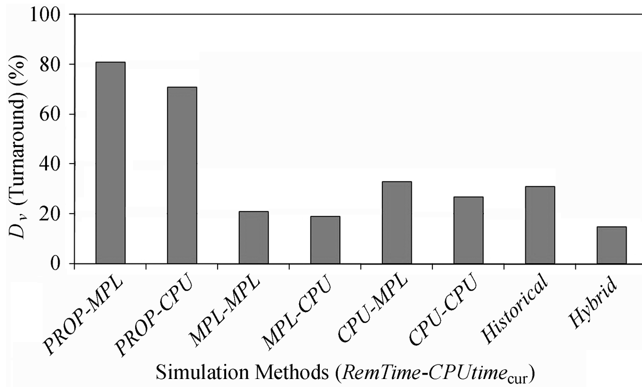

Fig.7. Turnaround error (%) applying a Basic policy.

From Fig.7, and considering the *RemTime* methods, we can see that the *Proportional* (denoted as PROP) method performs badly. This is due to the assumption that the future behaves like the past. This assumption is not true when the environment state changes continuously due to the local and parallel loads presents in a non-dedicated environment. In line with these results, we discarded the Proportional method for the following tests. Likewise, from the same Fig.7, we can see that $CPUtime_{\mathrm{cur}}$ estimation via *CPU* usage (denoted as CPU) is more reliable than estimation through the *MPL* (denoted as MPL) method. This happens because the *CPU* method represents reality better by considering the real percentage of *CPU* consumed by each task, while the *MPL* method assumes that every task consumes the same percentage of *CPU*. Likewise, the poor performance achieved by the Historical method, which is the most widely used to estimate the execution time in the literature, is surprising. This is due to the correlation of two reasons. On the one hand, the waiting queue length increases due to the restriction of the MPL_{paral} ($MPL_{\mathrm{paral}} = 1$) used by the Basic policy and, on the other hand, the historical repository, used by the Historical method, does not contain enough information to follow the evolution of a job in the waiting queue. As a consequence, the estimation error increases with the waiting queue length and the turnaround time prediction worsens. Finally, the good behavior of the Hybrid case is worth pointing out. This is analyzed in the following tests.

Next, we increased the complexity of the environment under study by increasing the MPL of parallel jobs to 4. In this context, we evaluated the sensitivity of the Simulated, Historical and Hybrid methods regarding the *NSP* (*Normal* and *Uniform*) and *JSP* (*Backfilling* and *FCFS*) scheduling policies. In relation to the *JSP* policies, as was expected, Fig.8 reflects that a backfilling policy is more unpredictable. This is due to the difficulty of tracking the variation in the order of the elements in the waiting queue. However, the results are almost always optimistic, due to the possibility of backfilling some of the waiting jobs, and hence reducing their waiting time. This means that the turnaround time is, in the worst case, underestimated. Regarding the *NSP* policies, we should distinguish between the Historical case and the others. For the Historical, we can see that the worst results are obtained using a conservative policy (*Uniform*) because it generates a longer queue length and as a consequence, the waiting time becomes more difficult to estimate. On the other hand, in the other cases we can see that a *Uniform* policy favors the predictability of the system over a *Normal* policy, because the former tries to balance the resources given to the parallel jobs. In such a case, the tasks forming a parallel job can evolve jointly, allowing the estimation methods

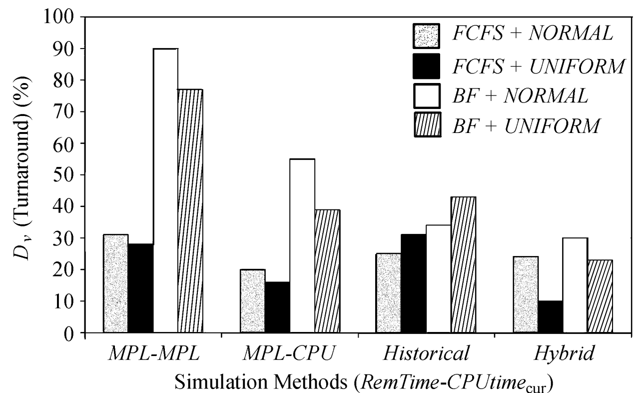

Fig.8. Turnaround error (%) vs. *JSP* and *NSP* scheduling policies.

to be more accurate. Focusing on the results obtained by the simulation methods (*MPL-MPL* and *MPL-CPU*), we can see that they follow the same trend obtained with the Basic policy shown in Fig.7. According to these results, the simulation method used in the rest of the tests explained in the following experimentation models the *RemTime* and $CPUtime_{\rm cur}$ based on *MPL* and *CPU* usage, respectively. Finally, it is worth pointing out that, for all the scheduling policies, a simulation system based on using a historical repository to compute the *RemainTime* (*Hybrid case*) achieves the best results, given that it combines the flexibility of a simulation system with the ability to represent a system as dynamic as our environment accurately.

The three estimation proposals (*Simulated*, *Historical* and *Hybrid*) described in this paper were compared with the two historical methods proposed by Li *et al.*[20] and Smith *et al.*[14] Smith *et al.* process the historical database by means of a weighted learning technique, where the relevance between data points and a query is determined with a heterogeneous Euclidean overlap metric and the estimation of the execution time is calculated by means of a kernel regression, where the kernel function is a Gaussian. On the other hand, Li *et al.* use a linear regression method (LR(5)) applied to different combinations of selected job attributes. Finally, the estimations produced by each set of attributes is averaged. In both cases, the job attributes used in our experimentation were the *user name*, *executable name* and *number of nodes allocated*. Note that these techniques were chosen due to the fact that they achieve better prediction results than other methods based on historical information[12-13]. Fig.9 shows the error achieved in the estimation of the turnaround time for all the methods compared. All these tests were done with a *Uniform NSP* policy and a *Backfilling JSP* policy. In general, we can see that our proposals based on a simulation engine (*MPL-CPU* and *Hybrid*) obtain much better results than Li and Smith's proposals, while our historical proposal is slightly better than them. In order to represent a system as complex as a non-dedicated cluster accurately, we have to collect

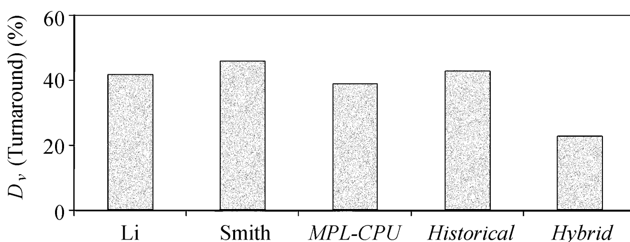

Fig.9. Comparison of our proposals in relation to Li and Smith's proposals.

more detailed information (cluster state, queue state and job characteristics) than the other historical systems orientated towards dedicated environments, as in the case of Li and Smith's proposals, which only store information about previously executed jobs together with their input parameters. Thus, we are able to follow the evolution of the cluster's state on each simulation step through the job's execution more accurately.

The next step consists of measuring the *waiting* and *execution* time deviations separately under the same conditions described above. Note that Li and Smith's results do not appear in Fig.10 because these authors do not discriminate between these two metrics, given that they give only results about average turnaround. In order to better show the relationship between the estimated and observed values for each metric and estimation method, the Fig.10 shows the deviation between the predicted and observed values by means of a linear regression model. Each scatter-plot represents the observed values (X) on the x-axis and the predicted ones (Y) on the y-axis. Moreover, these figures show the *normalized average of the absolute prediction error* (D_v), defined in (13), as well as the *adjusted coefficient of determination* (R_a). R_a measures how well the fitted linear model explains the reality of the system behavior. The R_a values range from 0 to 1, where $R_a = 1$ means that the model explains perfectly the estimated values with respect to the observed ones, and $R_a = 0$ otherwise. In general, from Fig.10, we can see that both methods based on simulation, *MPL-CPU* and *Hybrid*, perform better for both metrics. Basically, this is due to the fact that the estimation of the execution and waiting time under the simulation engine is based on the real state of the waiting queue and nodes at the beginning of each new simulation step and not on an estimated state, as in the Historical case. Likewise, we can see as the waiting time is more difficult to predict than the execution time, which is reflected in the lower values for R_a and higher values for D_v. An interesting case is the *Historical* one, where there is no correlation at all between the results achieved by both metrics. The high deviation of the waiting time is due to the high dynamism of the applied scheduling policies, such as the backfilling one, which provokes a wide range of different feasible states of the waiting queue for the same application. On the other hand, the estimation of the execution time is much better, due to the low variability among the different possible execution environments. We must take into account that, although the behavior of a single local user is unpredictable, the range of conduct of a group of local users, as for instance in a laboratory associated with practices on a specific subject, is much more limited.

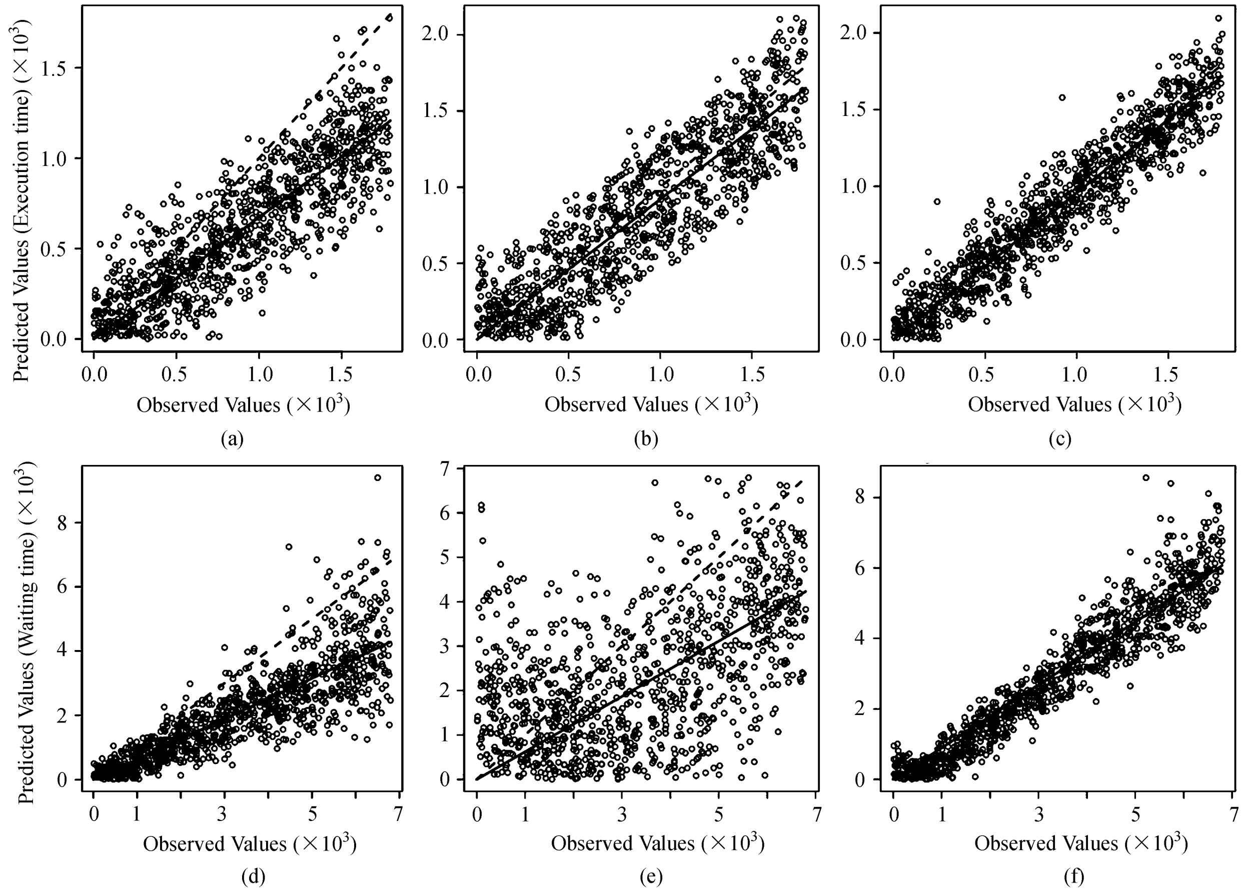

Fig.10. Linear regression analysis for execution ((a)∼(c)) and waiting ((d)∼(f)) time predictions. (a) *MPL-CPU*. (b) *Historical*. (c) *Hybrid*. (d) *MPL-CPU*. (e) *Historical*. (f) *Hybrid*.

Table 4. Time Cost of Our Estimation Proposals

	Simulation (*MPL-CPU*)	*Historical*	*Hybrid*
Time Cost (milliseconds)	7	19	98
Complexity	$O(N_{DRQ}^2 \times S_{cl})$	$O(N_{HistDB})$	$O(N_{DRQ}^2 \times N_{HistDB} \times S_{cl})$

6.1.3 Time Cost/Complexity Analysis

In this subsection, the time cost used by our proposals to estimate the turnaround time of a single job is measured. This cost has been compared with the complexity associated with each algorithm. Both are shown in Table 4.

Taking complexity into account, the low time cost of the Simulation case in relation to the Historical one could be surprising. This is due to the fact that the complexity of the Simulation algorithm is quadratic with the length of the running queue (N_{DRQ}), which is usually no longer than 5 jobs, whereas the order of the Historical one is linear with the number of elements in the Historical Repository (N_{HistDB}), which is at least 4 orders of magnitude greater than N_{DRQ}. Likewise, note that the time cost of the Hybrid one is strongly correlated to its complexity. Finally, we emphasize that the time cost is lower than hundred of milliseconds in all the cases, meaning that this is at least two orders of magnitude lower than the execution time of the parallel jobs (minutes). In this way, it is worth pointing out that the Li[20] and Smith's[21] estimation methods, used in this work for comparison purposes, have the same magnitude of time cost as we do. Although we need to take into account a more complex information for each job than these cited works, we can use a simpler process to search for similar jobs to the current job under estimation in the historical database.

6.2 Simulated Environment

In this subsection, we evaluate the performance of the off-line simulator by means of simulating the same cluster environment and workload used in the previous subsection. Thus, we compared this new set of

simulated results in relation to those obtained in the real cluster.

Regarding the parallel workload, as in the previous experimentation, we simulated a list of 1000 PVM/MPI NAS parallel jobs (CG, IS, MG, BT) with a size of 4, 8 or 16 tasks. Table 3 shows the characterization of these benchmarks used in the simulation. The local workload was modeled according to the values shown in Table 2.

In relation to the cluster, we simulated the same cluster used in the real environment. It is a homogeneous cluster made up of 64 nodes, where each node was modeled with a memory of 1 GB and a CPU of 5980 BogoMips.

6.2.1 *Accuracy of the Off-Line Simulator*

First, we analyzed the influence of the local load over the simulated (*Off-Line*) and the real (*On-Line*) cluster. In both environments, the same test was repeated, changing the estimation kernel from the *Simulation* method to the *Hybrid* one. In order to isolate the prediction from the difficulty of the scheduling policy, all the tests were tested using the easiest policy (*FCFS* + *Uniform*) policy. In addition, we vary the number of nodes with local load from 0% to 100%. Fig.11 shows the turnaround error for both scenarios. From Fig.11, we can see that the error performed by the on-line simulation is lower than the off-line in all the cases. Basically, this is due to the fact that the on-line estimation takes the real state of the cluster at each simulation step and so is able to correct dynamically the little error produced at each simulation step. On the other hand, in the off-line simulation case, the real state is only known at the beginning of the simulation process. Thus, the error produced at each simulation step is accumulated across the complete simulation process. However, this off-line error is always lower than 40% for the *Hybrid* one and 50% for the *Simulation* one. The second effect to note is the increment in the accuracy of the estimation when the local load increases, excluding the case of 0% local load in the nodes. This might seem contradictory, but is in fact perfectly understandable because when the local load increases, the available resources decrease, as do the opportunities to choose free nodes for scheduling jobs. As explained above, this situation favors the accuracy of the estimation. In the case of a small local load, lower than 25%, we can see as the accuracy is lower than 30% for all the cases, although the trend is exactly the same than the rest of analyzed loads. Finally, it is worth remarking that although the *Simulation* kernel obtains worse results than the *Hybrid* one, it follows exactly the same trends obtained with the latter.

Finally, for both kernels (*Simulation* and *Hybrid*), we

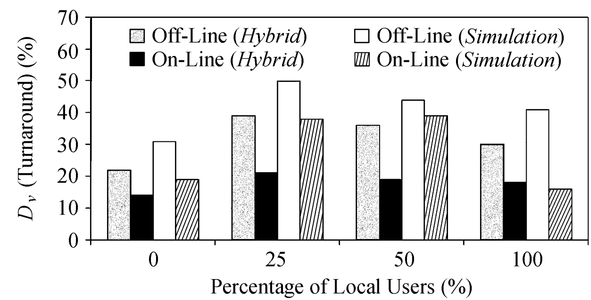

Fig.11. Turnaround error of the off/on-line simulator vs. different local loads.

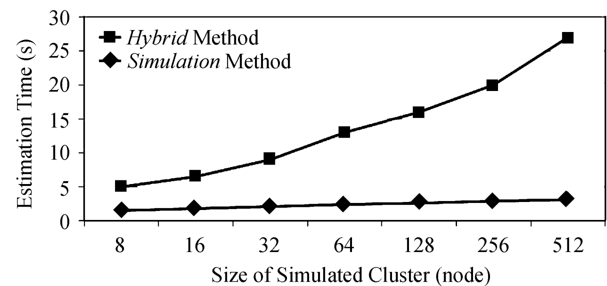

Fig.12. Estimation time with the off-line simulator.

measured the estimation time spent by the off-line simulator when simulating the complete parallel workload over a cluster environment, whose size was scaled down from 8 to 512 nodes. As was expected, the kernel based on the analytical methods (*Simulation*) obtains a much lower temporal cost than the *Hybrid* one. In fact, the temporal cost associated with the analytical methods is practically independent of the size of the cluster and waiting queue. On the other hand, the methods based on a historical repository (*Hybrid*) are totally dependent on both values. However, it is worth pointing out that this estimation cost is always kept to the order of seconds, which is two orders of magnitude lower than the makespan obtained in the real cluster.

7 Conclusions and Future Work

This paper proposes a prediction engine applicable to non-dedicated environments which is able to estimate the turnaround time for parallel applications, even in the presence of local loads uncontrolled by the parallel system. Our prediction system can be configured to use three different estimation methods: a *Historical* system that records past execution and tries to infer the future as a replication of the past, a *Simulation* system based on analytical models to estimate the *Remaining Execution time* and used *CPU time* for a given running application and a defined cluster state, and a system, named *Hybrid*, that merges a simulator engine for the scheduling process with a central execution time estimation system based on historical schemes. This

prediction engine was implemented in a scheduling system, named CISNE[8], which has the particular characteristic of being able to use any of the estimation methods in an *on-line* way, which means that a new estimation is made on every job arrival to a real cluster, or in an *off-line* mode, which estimates the behavior of the parallel workload over the simulated environments. It allowed our proposals to be evaluated in two different non-dedicated environments, a real and a simulated cluster.

In both environments, we observed that our *Hybrid* system gives the best results because it combines the flexibility of a simulation system together with the ability, given by the historical information, to represent accurately a system as dynamic as our environment. In fact, the turnaround error achieved by the *Hybrid* is always lower than 30%. Likewise, our estimation proposals were analyzed in relation to different job scheduling policies. We concluded that those policies including a backfilling scheme are inherently more difficult to estimate accurately due to the possibility of altering the job ordering in the queue. However, this estimation always tends to be optimistic. Another effect that was observed was the influence of the job distribution on the estimation process. For those policies that balance the resources it is easier to generate a more accurate estimation. Finally, we analyzed the influence of the local load on the estimation methods. Our experimental results showed an improvement in the accuracy of the estimation when the local load increased. This is due to the fact that when the local load increases, the available resources decrease, as do the opportunities to backfill a job.

In the future, we wish to extend our system to include a couple of extra capacities. On one hand, we want to apply the system to a *multicluster* environment. It means taking the network and heterogeneity (memory and CPUs) into account. On the other hand, we want to support *soft-real-time* applications from both the local and parallel user points of view.

References

[1] Acharya A, Setia S. Availability and utility of idle memory in workstation clusters. In *Proc. the ACM SIGMETRICS/PERFORMANCE 1999*, Atlanta, USA, May 1-4, 1999, pp.35–46.

[2] Kuo C H. A study of resource allocation for non-dedicated distributed shared memory systems [M.S. Thesis]. "National Cheng-Kung University", 2004.

[3] Mahanti J, Eager D L. Adaptive data parallel computing on workstation clusters. *Journal of Parallel and Distributed Computing*, 2004, 64(11): 1241-1255.

[4] Stava M, Tvrdik P. Overlapping non-dedicated clusters architecture. In *Proc. Int. Conf. Computer Engineering and Technology*, Singapore, Jan. 22-24, 2009, pp.3-10.

[5] Litzkow M, Livny M, Mutka M. Condor — A hunter of idle workstations. In *Proc. the 8th Int. Conference of Distributed Computing Systems*, San Jose, USA, Jun. 13-17, 1988, pp.104-111.

[6] Chowdhury A, Nicklas L, Setia S, White E. Supporting dynamic space-sharing on non-dedicated clusters of workstations. In *Proc. the 17th International Conference on Distributed Computing Systems* (*ICDCS 1997*), Baltimore, USA, May 27-30, 1997, pp.149-158.

[7] Goscinski A M, Wong A. A study of the concurrent execution of parallel and sequential applications on a non-dedicated cluster. *Parallel Computing*, 2008, 34(2): 69-91.

[8] Hanzich M, Giné F, Hernández P, Solsona F, Luque E. CISNE: A new integral approach for scheduling parallel applications on non-dedicated clusters. In *Proc. EuroPar 2005*, Lisbon, Portugal, Aug. 30-Sept. 2, 2005, pp.220-230.

[9] Urgaonkar B, Shenoy P. Sharc: Managing CPU and networks bandwidth in shared clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2004, 15(1): 2–17.

[10] Harchol-Balter M, Li C, Osogami T, Scheller-Wolf A, Squillante M S. Cycle stealing under immediate dispatch task assignment. In *Proc. the 15th Annual ACM Symp. Parallel Algorithms and Architectures*, San Diego, USA, Jun. 7-9, 2003, pp.274-285.

[11] Lafreniere B J, Sodan A C. Scopred — Scalable user-directed performance prediction using complexity modeling and historical data. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, Cambridge, USA, Jun. 19, 2005, pp.62-90.

[12] Downey A B. Predicting queue times on space-sharing parallel computers. In *Proc. the 11th International Symposium on Parallel Processing* (*IPPS 1997*), San Juan, Puerto Rico, Apr. 12-16, 1997, pp.209-218.

[13] Gibbons R. A historical application profiler for use by parallel schedulers. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, Geneva, Switzerland, Apr. 5, 1997, pp.58-77.

[14] Smith W, Foster I, Taylor V. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 2004, 64: 1007-1016.

[15] Wolski R. Experiences with Predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 2003, 30(4): 41-49.

[16] Yang L, Schopf J M, Foster I. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *Proc. Supercomputing*, Phoenix, USA, Nov. 15-21, 2003, pp.262-273.

[17] Kerbyson D J, Harper J S, Craig A, Nudd G R. PACE: A toolset to investigate and predict performance in parallel systems. In *Proc. European Parallel Tools Meeting*, Onera, France, Oct. 23, 1996.

[18] Jarvis S A, Spoone D Pr, H N Lim Choi Keung, Cao J, Saini S, Nudd G R. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems Special Issue on System Performance Analysis and Evaluation*, 2004, 22(7): 745-754.

[19] Hanzich M, Hernandez P, Luque E, Gine F, F Solsona, Lerida J L. Using simulation, historical and hybrid estimation systems for enhancing job scheduling on NOWs. In *Proc. IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 25-28, 2006, pp.1-12.

[20] Li H, Groep D, Templon J, Wolters L. Predicting job start times on clusters. In *Proc. the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid* (*CCGrid 2004*), Chicago, USA, Apr. 19-22, 2004, pp.301-308.

[21] Smith W, Wong P. Resource selection using execution and queue wait time predictions. *NAS Technical Reports*, 2002.

[22] Mu'alem A W, Feitelson D G. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transaction on Parallel & Distributed Systems*, 2001, 12(6): 529-543.

[23] Nissimov A, Feitelson D G. Probabilistic backfilling. In *Proc. JSSPP 2007*, Seattle, USA, Jun. 17, 2007, pp.102-115.

[24] Zhang Y, Franke H, Moreira J E, Sivasubramaniam A. An integrated approach to parallel scheduling using gang-scheduling, backfilling and migration. *IEEE Transactions on Parallel and Distributed Systems*, 2003, 14(3): 236-247.

[25] Talby D, Feitelson D G. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Proc. the 19th IEEE Int. Parallel and Distributed Processing Symposium* (*IPDPS 2005*), Denver, USA, Apr. 4-8, 2005.

[26] Tsafrir D, Etsion Y, Feitelson D G. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, June 2007, 18(6): 789-803.

[27] He L, Jarvis S A, Spooner D P, Nudd G R. Dynamic, capability-driven scheduling of dag-based real-time jobs in heterogeneous clusters. *International Journal of High Performance Computing and Networking*, 2004, 2(2-4): 165-177.

[28] Dinda P A. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 2006, 17(2): 160-173.

[29] Lin B, Sundarara A I, Dinda P A. Time-sharing parallel applications with performance isolation and control. In *Proc. International Conference on Autonomic Computing*, Jouksonville, USA, Jun. 11-15, 2007, p.28.

[30] Brevik J, Nurmi D, Wolski R. Using model-based clustering to improve predictions for queueing delay on parallel machines. *Parallel Processing Letters* (*PPL*), Jan. 2007, 17(1): 21-46.

[31] Shmueli E, Feitelson D G. Backfilling with lookahead to optimize the performance of parallel job scheduling. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, USA, Jun. 24, 2003, pp.228-251.

[32] Srinivasan S, Kettimuthu R, Subrarnani V, Sadayappan P. Characterization of backfilling strategies for parallel job scheduling. In *Proc. International Conference on Parallel Processing Workshops* (*ICPPW 2002*), Vancouver, Canada, Aug. 20-23, 2002, pp.514-522.

[33] Arpaci R H, Dusseau A C, Vahdat A M, Liu L T, Anderson T E, Patterson D A. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. the ACM SIGMETRICS/PERFORMANCE 1995*, 1995, pp.267-277.

[34] Giné F, Solsona F, Hanzich M, Hernández P, Luque E. Cooperating coscheduling: A coscheduling proposal aimed at mon-Dedicated heterogeneous NOWs. *Journal of Computer Science and Technology*, 2007, 22(5): 695-710.

[35] Hanzich M, Giné F, Hernández P, Solsona F, Luque E. A space and time sharing scheduling approach for PVM non-dedicated clusters. In *Proc. EuroPVM/MPI 2005*, Sorrento, Italy, Sept. 18-21, 2005, pp.379-387.

[36] Mutka M, Livny M. The available capacity of a privately owned workstation environment. *J. Performance Evaluation*, 1991, 12(4): 269-284.

[37] Bailey D H, Barszcz E, Barton J T, Browning D S, Carter R L, Dagum D, Fatoohi R A, Frederickson P O, Lasinski T A, Schreiber R S, Simon H D, Venkatakrishnan V, Weeratunga S K. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 1991, 5(3): 63-73.

[38] Lublin U, Feitelson D G. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 2003, 63(11): 1105-1122.

Mauricio Hanzich received his B.S. degree from the Universidad Nacional of Comahue, Argentina, and his M.S and Ph.D degrees in computer science from the Universitat Autònoma of Barcelona (UAB), Spain, in 2002, 2004 and 2006 respectively. He is currently a researcher at the Barcelona Supercomputing Center. His research interests include cluster scheduling and parallel computing.

Porfidio Hernández received the B.S., M.S. and Ph.D. degrees in computer science from UAB, Spain, in 1984, 1986 and 1991, respectively. He is currently an associate professor of operating systems at UAB. His research interests include operating systems, cluster computing and multimedia systems.

Francesc Giné received the B.S. degree in telecommunication engineering from the Universitat Politecnica de Catalunya (UPC), Spain, in 1993 and the M.S. and Ph.D degrees in computer science from UAB, Spain, in 1999 and 2004, respectively. He is currently an associate professor of computer arquitecture at University of Lleida (UdL), Spain. His research interests include cluster, multicluster and peer-to-peer computing and scheduling-mapping for parallel processing.

Francesc Solsona received the B.S., M.S. and Ph.D. degrees in computer science from the UAB, Spain, in 1991, 1994 and 2002 respectively. Currently, he is an associate professor and chairman of the Department of Computer Science at the UdL, Spain. His research interests include distributed processing, cluster, multicluster and peer-to-peer computing, and administration and monitoring tools for distributed systems.

Josep L. Lèrida obtained his B.S. degree in computer science from the Open University of Catalonia (OUC) in 2004 and his M.S. and Ph.D. degrees from UAB in 2006 and 2009, respectively. He is currently an associate professor of the Computer Science Department at UdL. His current research interests include performance modeling, prediction and scheduling in cluster, grid and P2P environments.