The XtreemOS JScheduler: Using Self-Scheduling Techniques in Large Computing Architectures

F. Guim
Barcelona Supercomputing Center
fguim@bsc.es

I. Rodero
Barcelona Supercomputing Center
irodero@bsc.es

M. Garcia
Barcelona Supercomputing Center
marta.garcia@bsc.es

J. Corbalan
Barcelona Supercomputing Center
julita.corbalan@bsc.es

Abstract

Several centralized scheduling solutions have been proposed in the literature for environments composed of several independent computational resources, such as centralized schedulers, centralized queues and global controllers. These approaches use a unique scheduling entity responsible for scheduling all jobs submitted by users. In our previous work we proposed the use of self-scheduling techniques to dispatch jobs which are submitted to a set of distributed computational hosts, which are in turn managed by independent schedulers (such as MOAB or LoadLeveler). In the ISIS-Dispatcher, scheduling decisions are made independently for each job instead of using a global policy where all jobs are considered.

In this paper we present how the ISIS-Dispatcher techniques can be used in the XtreemOS architecture for manage the jobs. This system is designed to be deployed in large scenarios that potentially involve thousands of resources. In such systems it is not feasible to make the dispatcher contact to all the systems. It is not realistic to suppose that the dispatcher stores the information about all the resources and where they are located. Obviously, this approach would imply problems of scalability. In this paper we also evaluate the impact about the amount of resource information that the dispatcher can collect during the job scheduling.

1 Introduction

The increasing complexity of local systems has led to new distributed architectures. These new systems are composed of multiple computational resources with different characteristics and policies. In these new distributed scenarios, traditional scheduling techniques have evolved into more complex and sophisticated approaches where other factors, such as the heterogeneity of the resources [9] or geographical distribution [7], have been taken into account.

Distributed HPC architectures are usually composed of several supercomputing centers containing many hosts. In the job scheduling strategies proposed in the literature, jobs are submitted to one centralized scheduler which is responsible for scheduling all the submitted jobs to all the computational resources available in the system. Thus, users submit jobs to a scheduler which schedules the jobs according to a global scheduling policy. The policy takes into account all the queued jobs and the resources available in the center to decide which jobs have to be submitted where, and in what order.

Following a similar approach of the AppLes project [2], in [10] we proposed replacing the use of a global scheduler or global structures to manage jobs submitted in these scenarios, with the use of self-scheduling techniques which dispatch the jobs that users want to submit to the set of distributed hosts. In this new architecture, jobs are scheduled by their own 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 other jobs that other users have submitted to the system. Thus, the job itself decides which is the most appropriate resource for its execution. The target architectures of our work are distributed 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, we propose 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.

In this paper we present how the ISIS-Dispatcher techniques can be used in the XtreemOS (henceforth referenced as XOS) architecture for schedule the submitted jobs that are submitted to its scheduling entities (*jSched-*

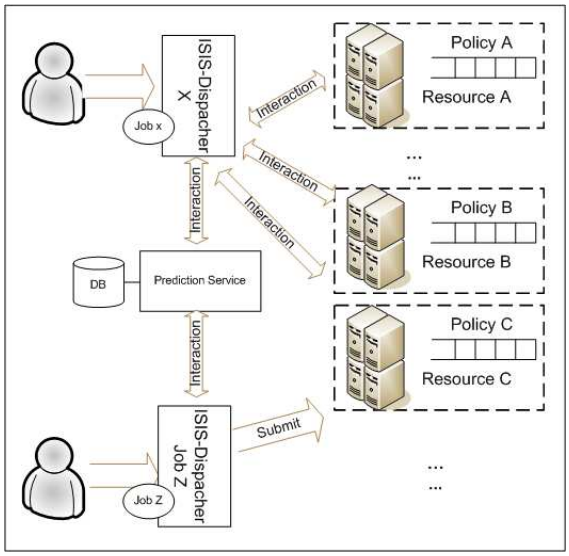

Figure 1: The Extended ISIS-Dispacher architecture

uler). The XOS system has been designed to be deployed in large scenarios that potentially involve thousands of resources. In such systems it is not feasible to make the dispatcher contact to all the systems. It is not realistic to suppose that the dispatcher stores the information about all the resources and where they are located. Obviously, this approach would imply problems of scalability.

The XOS Application Discovering System (henceforth referenced as ADS) component is the responsible to provide to the *jScheduler* the information regarding the resource that satisfy the job requirements. In this paper, we evaluate the impact of the amount of resource information the dispatcher receives from the ADS during the job scheduling. Furthermore, in the evaluation presented, we evaluate our approaches in a possible XtreemOS architecture: a large computing system composed by 200 different resources containing more than 41000 processors; the computational resources are heterogeneous, with different local policies and different sizes (form 2 processors until 25000 processors). We demonstrate how our techniques can be successfully used in this big and heterogeneous systems.

The rest of the paper is organized as follows: section 2 presents the related work; next the XtreemOS architecture is explained; section 4 describes how the ISIS-Dispatcher algorithm can be included in the architecture, and 5 describes how it has been modeled in the Alvio simulator; next we present the experiments that we designed for evaluated the presented techniques; and finally sections 7 and 8 present the evaluation and conclusions.

2 Motivation and related work

In the new large distributed systems, as with grids, more global scheduling approaches are being considered. In [25], Yue proposes using a global backfilling policy within a set of independent hosts where each of them is managed by an independent scheduler. The core idea of Yue's algorithm is that the user submits a job to a specific system, managed by an independent scheduler and a global controller tries to find out if the job can be backfilled to another host of the center. Sabin et al. studied in [9] the scheduling of parallel jobs in a heterogeneous multi-site environment. They propose carrying out a global scheduling within a set of different sites using a global meta-scheduler where users submit their jobs. They compare the effect of submitting jobs to the site with the least instantaneous load to the effect of submitting jobs to K different sites and, once the job is started in one site, cancelling the rest of the submissions.

In [7] the authors analyze the impact of geographical distribution of Grid resources on machine utilization and the average response time. A centralized Grid dispatcher which controls all resource allocations is used. The local schedulers are only responsible for starting the jobs after resource selection has been made by the Grid Scheduler. A similar approach is presented by Schroeder et al. in [19], where they evaluate a set of task assignment policies using the same scenario (one central dispatcher).

In [18] Pinchak et al. describe a metaqueue system which manages the jobs with explicit workflow dependencies, and involves the use of a centralized scheduling system. However the submission approach is different from the one discussed above. Here the system is composed of a user-level metaqueue that interacts with the local schedulers. In this scenario, instead of the push model, where jobs are submitted from the metaqueue to the schedulers, place holding is based on the pull model, where jobs are dynamically bound to the local queues on demand.

All the previous enumerated solutions for distributed architectures are based on the use of centralized scheduling systems where jobs are submitted. The scheduling policies are centralized. In the AppLes project [2][1] Berman et al. introduced the concept of application-centric scheduling, where everything about the system is evaluated in terms of its impact on the application. Each application developer schedules his/her application so as to optimize his/her 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 [10] and [11] we presented the non-centralized job

oriented scheduling policy implemented by the ISIS-Dispatcher. It is a scheduling entity that is assigned to one, and only one, job. As a result, when a user wants to submit a job, a new dispatcher is instantiated. This new dispatcher is responsible for submitting the job to the computational resource that best satisfies the job requirements, and which maximizes its response time or other job metrics. The ISIS-Dispatcher is designed for deployment in large systems, for example, groups of HPC Research Centers or groups of universities. The core of the ISIS-Dispatcher algorithm is based on task selection policies. In the referenced paper we demonstrated how the new task selection policy (Less-WaitTime) based on the wait time prediction achieved higher performance than previously proposed policies. The main advantage of the new policy is that it takes into account the capacity of the available resources and scheduling information while the others do not (i.e: Less-WorkLeft, Less-Queued-Jobs etc. [19][14]). In [11] paper we proposed the use of prediction techniques in the ISIS-Dispatcher architecture as an alternative to the user estimates which we initially proposed. We described a new evolutive prediction model based on classification trees and discretization techniques designed for these distributed environments. We introduced the modelling of the economic cost of using different computational resources. Furthermore, we proposed 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.

2.1 Data mining and prediction techniques

As will be described later, the prediction system provides the dispatcher with estimates which allow it to make more intelligent scheduling decisions. In [11] we described a new evolutive prediction model based on classification trees and discretization techniques designed for these distributed environments. The prediction model that we used in our experiments was developed using the Weka [15] framework. Its goal was 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 the paper we described how the model is built during a simulation and we provided its evaluation. We shown how the described 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. In the evaluation presented in this paper we present we have used the same prediction system.

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 [6][21][22]. A relevant study in this area is the prediction work presented by Lui in [16], where the use of the K-Nearest-Neighbor algorithm is used to predict queue wait times. What we proposed in [10] was to use wait time predictions based on scheduler reservation tables. 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.

3 The XtreemOS Project

The emergence of Grids enables the sharing of a wide range of resources to solve large-scale computational and data intensive problems in science, engineering and commerce. While much has been done to build Grid middleware on top of existing operating systems, little has been done to extend the underlying operating systems to enablee and facilitate Grid computing, for example by embedding important functionalities directly into the operating system kernel.

The XtreemOS [17] (henceforth referenced as XOS) project aims at investigating and proposing new services that should be added to current operating systems to build large Grid infrastructure in a simple way. XtreemOS targets the Linux well-accepted open source operating system extending it to Grid with native support for virtual organizations. One of the most important challenges in XtreemOS is the identification of the basic functionalities which are to be embedded in the Linux kernel.

A set of operating system services, extending those found in the standard Linux distribution, will provide Linux users with all the Grid capabilities associated with current Grid middleware, but fully integrated into the OS. The underlying Linux OS will be extended as needed to support virtual organizations spanning across many machines and to supply appropriate interfaces to Grid OS services.

3.1 The XtreemOS Scheduling Architecture

The Application Execution Management (AEM) [23] is the component of the XOS responsible of job and resource management. It is mainly divided into two parts: the user or client side and the server or system side. The user side basically offers the API to access the AEM services. The user/client side includes the implementation of the command line and third party libraries such as SAGA to be built on top of the Basic XtreemOS API. This layer mainly pre-process arguments, calls the system side and post-process results.

The system side is mainly composed of the XOSD (XtreemOS Daemons) and some distributed services such as the JobDirectory and the jResMatching (for more information see [23]). Each node (also known as computational resource) of the system has its own XOSD. It contains a subset of the total amount of information about jobs actives in the VO (concerning execution management). Each XOSD manages information about jobs submitted in the node, running in the node, and about the node itself (one node is a resource). The main services it provides are:

- jController holds the job information and its three main goals are first, to ensure that the scheduling agreement between the job and the resources is accomplished, second, to validate the job is executing as expected, and third, to act as a gateway for the job. The jController holds most of the information associated to the job and it is the only service that has a global vision of it. It is the service that provides self-management to XtreemOS jobs.

- jScheduler schedules one job. The jScheduler receives a pre-selection of resources from the jResMatching (resource matching global service used to store and gather resource information) based on job resource requirements and in a second step it performs a negotiation with pre-selected resources in order to decide the final allocation. The jScheduler service is stateless and from one job scheduling to a next one no status is stored.

- jMonitor collects information from all the processes of the job and adds them in a job basis. One of the goals of the jMonitor is to provide a monitoring service as powerful and flexible as possible, allowing advanced versions of XtreemOS to add new metrics without changing either the API or the system architecture.

- jExecMng is a distributed service that implements methods for managing the execution of the job.

In the current architecture (Figure 2), there are two components that are used by the AEM. Given a job description, the *ADS* component provides to the *jScheduler* lists of *K* components that satisfy such requirements and where the job can be submitted. The *ResourceManager* is the responsible to provide information to the other components concerning the host in which they are running. Thus, the *jScheduler*, contacts to this component to gather the information about the resources that can be used by the job.

As its described in the figure 2, when a job (henceforth JOB) with a set of requirements (henceforth REQS) and characteristics (henceforth CHARS) is submitted to the XOS system, the following steps are carried out:

1. A *jScheduler* is created for schedule the job submission. (1)

2. The *jScheduler* contacts to the *ADS* and requires a resource that satisfy the job requirements and characteristics. The *ADS* returns the reference to *ResourceManager* that manages the resource that it has selected. (2)

3. If the reference is not null, the *jScheduler* contacts to the the *ResourceManager* and submits the job.

Something important to is that the ADS abstracts to the *jScheduler* about the complexity of the architecture that is behind him. It is an specialized component that has been designed for efficiently provide information about parts of huge systems (potentially composed by millions of resources) [5].

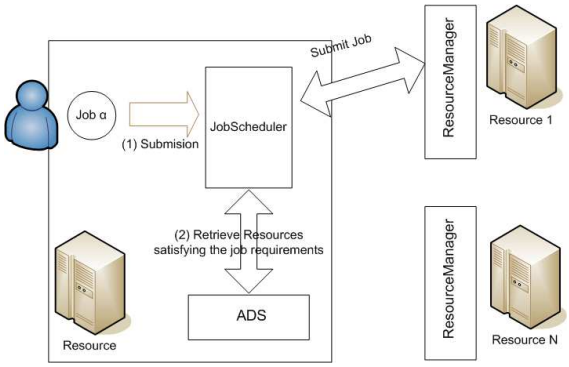

Figure 2: The current XtreemOS Architecture

4 Including Self-Scheduling techniques in XreemOS Scheduling Architecture

To include the ISIS techniques in the XtreemOS infrastructure very few changes have to be considered. As can

be observed in the Figure 3 there is only one new element that could be considered in the submission system: the prediction system. This component, given the characterization of the submitted job and a given resource of the system, it has to estimate the runtime that this job is expected to experiment in such resource.

Similar as has been doing until the moment, the submission to the system is done using the *jManager*. When the user wants to submit a job (henceforth JOB) with a set of requirements (henceforth REQS) and characteristics (henceforth CHARS) the JobManager will still create the *jScheduler* that will decide where the job has to be submitted. However, in this extension the user will be able to specify which job metric (henceforth METRIC) has to be optimized in during the scheduling process (the available metrics are described in the last section of this document). In the case that no metric is specified a default metric will be used. The *jScheduler* will follow the next algorithm:

1. The *jScheduler* contacts to the appropriate *ADS* instance and it request for K resources $\{resource1,..,resourceK\}$ that satisfy the job REQS and CHARS.
2. For each resource in $\{resource1,..,resourceK\}$ the *jScheduler*:
   (a) It will contact to the prediction system and will require the *EstimatedRuntime* for the JOB in the resource. The *jScheduler* will provide both job requirements and job characteristics. (This runtime estimation can be also considered like local resource specific information. It is runtime in a specific resource).
   (b) If local resource specific information, such as the estimated wait time, is required for compute the metric METRIC that is being to be optimized, the *jScheduler* will contact to the appropriate Resource Manager that manages the given resource and will gather such information. For some metrics, the *jScheduler* will have to provide a set of data. (described below).
   (c) It will compute the METRIC_resource using all the collected information.
3. It will select the resource that optimizes the METRIC. This optimization function is potentially different for each of the available metrics. Usually, it will be the minimum function (i.e: for selecting the center with least estimated wait time).

One of the objectives of this paper is to evaluate the impact of the value K used in the step *1* of the dispatching process. Obviously, as higher K values are used, the space of search in which the dispatcher is able to retrieve information is higher. Thus, the quality of the scheduling is expected to be higher. However, using K with high values implies problems of scalability and performance. In the evaluation section we provide and study that shows the impact of this value in the scheduling.

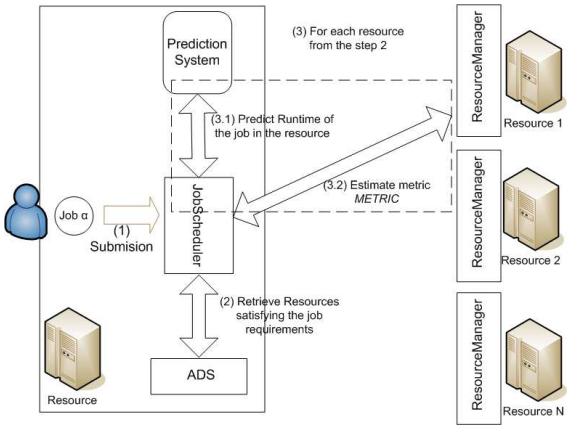

Figure 3: Including the ISIS-Dispatcher in the XreemOS Architecture

5 The Alvio Simulator

All our experiments were conducted using the C++ event-driven Alvio simulator. The simulator models the different components which 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: First-Fit, First-Continuous-Fit and LessConsume policies [13]), and the computational resource model [12]. A simulation allows us to simulate a given policy with a given architecture. Currently, three different policies have been modeled: the First-Come-First-Served policy, the Backfilling policy, and finally, the ISIS-Dispatcher scheduling policy and the RUA-Backfilling. For 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 main simulation component is the simulator engine, which manages all the simulation events and instantiates all the events of the simulation. In this model, when a *JOB_ARRIVAL* is triggered, a new *jScheduler* entity is created for the job. First, it discovers the different local systems that are available to it using the *ADS*. In this new version of the system, the *jScheduler* decides which center to submit the job by gathering the scheduling information for the local resource from the different

ResourceManagers and the runtime estimations from the prediction system. This last component includes a historical database containing all the information about the performance variables of jobs which have finished in the system, and a set of predictors that estimate the job performance variables using different techniques (i.e: statistical techniques or dataminig techniques). Once the *jScheduler* has chosen where to submit the job it contacts the scheduling component which manage the resource, and submits the job. At this point, the job becomes scheduled at *the local scenario*. As we have already outlined in our previous works, this component manages and allocates jobs based on the reservation table which models how the computational resources are used (processors, memory etc.). Once the job has been executed, the scheduling component provides information regarding job execution to the *jScheduler* which is managing its execution. The dispatcher contacts the prediction service and provides feedback about the job execution. The prediction service updates its historical database and may update the prediction models that it is using.

For the evaluations presented in this paper, dispatcher behaves as the *jScheduler* and it knows the available scheduling resources that are available in the system by contacting to the *ADS*. This new component allows the dispatcher to dynamically gather the information about the resources that are available at a given point of time. In the simulator configuration we can specify the amount of resources that the *jScheduler* requires to the *ADS* to perform the scheduling.

In the presented model, when a job is submitted to the system, it is assigned randomly to one of the *ExecutionManager*. This manager instantiates the corresponding *jScheduler* for the job. The *ADS* component is instantiated at the beginning of the simulation. When the *jScheduler* ask for K resources that satisfies the job requirements, the ADS selects the K requested centers using a random variable that follows a normal distribution $Center_{id}\ U[0,n]$ (where n is the number of known centers for the *ADS*).

6 Experiments

In this section we present the metrics, the workloads used in the simulations and the scenarios that were designed to evaluate the proposal.

6.1 The task asignament policies

As we have already introduced, the metrics that are used in the core of the ISIS-Dispatcher algorithm are based on task selection policies. In this paper we evaluate two different task assignment policies:

- The Less-WaitTime-pred (LWT-pred) policy minimizes wait time for the job using the predicted runtime provided by the prediction system. Given a static description of a job, including the runtime prediction, the local resource provides the estimated wait time for the job based on the current resource state. As in [10], 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 (see below).

- The Less-SlowDown-pred (LSLD-pred) 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.

In [10] 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 [11] we proposed the two policies used in this evaluation. They shown how the runtime prediction can also provide good results rather than using the user estimates.

6.2 Workloads

For the evaluations presented in this paper we have used a set of workloads that we consider representative of the current HPC infrastructures and that are evaluated. Note they go from small cluster composed by two processors until large and power computational resources with 9120 processors. More information about their properties and characteristics can be found in the workload archive of Dror Feitelson [8][3]. The workloads used are:

1. The NASA Ames iPSC/860 log.
2. The Los Alamos National Lab (LANL-CM5) log.
3. The San-Diego Supercomputer Center Paragon (SDSC-Par).
4. The Cornell Theory Center (CTC) SP2 log.
5. The Lawrence Livermore National Lab (LLNL).
6. The Swedish Royal Institute of Technology (KTH) IBM SP2 log.
7. The San Diego Supercomputer Center (SDSC-SP2) SP2 log.
8. The LANL Origin 2000 Cluster (Nirvana) log.
9. The OSC Linux Cluster log (OSC).
10. The San Diego Supercomputer Center Blue Horizon log (SDSC-Blue).
11. The HPC2N log.

12. The DAS2 5-Cluster Grid Logs.
13. The San Diego Supercomputer Center DataStar log (SDSC-DataStar).
14. The LPC Log.
15. The LCG Grid log.
16. The SHARCNET log .
17. The LLNL Atlas log.
18. The LLNL Thunder log.

The trace that was used in the evaluation was generated by fusing the first eight months of each trace. We chose these workloads because they contained jobs with different levels of parallelism and with run times that varied substantially.

6.3 The Scenarios

6.3.1 The Local system

We have already emphasized that the goal of this work was to evaluate the viability of using the ISIS-Dispatching techniques in large HPC infrastructures. To do this, we have used more than 17 traces collected from many HPC and Grid architectures (introduced in the previous subsection). Furthermore, we have defined a possible large multi-syte HPC System composed by all the different resources that are related to the traces. The architecture that we have evaluated is presented in the table 1. The system is composed by 200 different computing elements and it has more than 40.000 processors. Each of the resources in the table has associated two different characteristics:

- The Job Scheduling Policy that is used in the local system. It can be Shortest-Job-Backfilled-First [24] (SJBF), EASY Backfilling [20], LXWF Backfilling [4], Shortest Job First (SJF) and First-Come-First-Served (FCFS).

- The CPU Factor (third column) is used to model the computational power of each the computational resources. When the job α with original runtime $\alpha_{\{OriginalRunTime,rt\}}$ (specified in the SWF trace used in the simulation) is submitted to the resource σ the runtime of the job in the center becomes $\alpha_{\{RunTime,rt\}} = \alpha_{\{OriginalRunTime,rt\}} * CPUFactor$.

The scenario defined for this evaluation shows a heterogeneous and large multi-site HPC system. Note, that the resources are are really different in terms of processors computational power (CPU Factors from 1 until 8), number of processors (from 2 until 25000) and scheduling policy (from the FCFS until SJBF).

6.3.2 The XtreemOS Architecture

In this paper we have evaluated the described system using the ISIS-Techniques in the *JScheduler*. As we have already introduced, for this paper we have also modeled the usage of the *ADS*. The different scenarios that we have evaluated are:

- The task assignment policies: The Less-WaitTime-pred (LWT-pred) and Less-SlowDown-pred (LSLD-pred).

- The number of resources that are requested by the *jScheduler* to the ADS for a given job. In each experiment we have evaluated the effect of asking to the ADS *K* times the number jobs of requested processors. We have evaluated all the values included in the interval $K \in [1, 55]$.

7 Evaluation

7.1 The original scenarios

Table 2 shows the performance that each of the centers has shown evaluating its workload with the properties presented in the table 1. The last row presents the average values considering all the centers. It shows the average and 95^{th} percentile for the slowdown and the wait time. As can be observed, the performance of the different systems radically differ. The performance of the scheduling policies and CPU factors used in each experiment has a direct impact in the performance values presented in the table.

The two centers that have worst performance are the *LANL-CM5*, *DAS2-fs4* and the *HPC2N*. They show slowdowns from 221 until 1364. The main reason is that they use the policy FCSC. On the other hand, the *DAS2-f0*, *LPC* and the *SDSC-SP2* shows the best performance, in terms of slowdown and wait time. These last configurations are using the SJBF policy which has been demonstrated in the literature that provides a very good performance result.

There is also a clear impact of the CPU factor to the performance of the systems. Thereby, the *Thunder*, *HPC2N* and *KTH-SP2* show bad performance values. Contrary, the *DAS-fs0*, *DAS-fs3* and *SDSC-Par* show good performance values. Also, these lasts onces are using scheduling policies that achieve good performance and they also have the best CPU factors.

I all the cases, the performance obtained when reducing the computational power and the policy is not surprising. For instance, using FCSC or reducing by four the computational power of a given center significantly increases its slowdown and wait time. Thereby, The capacity of the resource of execute the same workload was

Center	CPUs	Fact.	Policy
NASA Ames	128	4	SJBF
LANL-CM5	1024	1	FCFS
SDSC Paragon	416	1	EASY
CTC IBM SP2	512	2	EASY
KTH	100	4	EASY
SDSC SP2	128	4	LXWF
Nirvana	2048	4	EASY
OSC	178	4	SJBF
SDSC-Blue	1024	2	FCFS
HPC2N	240	4	EASY
DAS-fs0	144	4	EASY
DAS-fs1	64	1	SJF
DAS-fs2	64	5	SJBF
DAS-fs3	64	1	SJBF
DAS-fs4	64	5	FCFS
SDSC-DS	184	5	
LPC	70 x 2	8	FCFS
LGC	100 x 250	5	EASY
Sharnet	6 x 128	5	SJF
	1 x 1068	5	SJF
	1 x 1536	5	SJF
	1 x 3072	5	SJF
	1 x 384	5	SJF
Atlas	1152	3	FCFS
Thunder	1024	8	EASY
CM5	1152	3	FCFS

Table 1: Centers Configuration

Metric:	Wait time		Slowdown	
Center	**Avg**	95_{th}	**Avg**	95_{th}
CTC-SP2	5249	29586	7,76	39,01
LCG	434,12	4320	4,3	23,32
DAS2-fs0	22,68	135	1,11	1,69
DAS2-fs1	5576	43414	11,71	21,18
DAS2-fs2	29594	99109	6,33	14,44
DAS2-fs3	4,52	100	1,03	3,23
DAS2-fs4	39053	192140	221	934,33
HPC2N	23980	87607	72,05	299,5
KTH-SP2	8864	54222	74,46	571,5
LANL-CM5	126565	308231	1364	4061
LPC	133	1323	1,23	3,42
Atlas	1993	14217	3,18	12,97
Thunder	18891	47758	138	366,8
BLUE	12383	27644	68,80	164,2
Par	453,12	12000	7,32	18,42
OSC	1233,32	25433	5,443	24,43
SDSC-SP2	116,12	1233	1,45	4,22
NASA	232,45	2133	2,43	10,43
Sharnet	649	4432	43,6	749
All	18198	29345	135,5	653

Table 2: Centers Performance

reduced four times. Thus, the original scenario cannot cope with the same job stream. The main concern to use these configurations was to evaluate later this configuration in the distributed scenario.

7.2 The XtreemOS Architecture

Figures 4 and 5 present the improvement of both average wait time and bounded slowdown against the original scenarios. In the X-Axes the graphic shows the K value that has been used in the experiment when carrying out the dispatching algorithm (see section 4). For instance, $K = 2$ means that the dispatcher has requested to the ADS *2* resources that satisfy the job requirements. The Y-Axes present the improvement of the presented variable respect the original scenario. In each figure we present the improvement of a given metric for the two task assignment policies presented in the subsection 6.1.

The bounded slowdown (Figure 4) starts improving respected the original scenario for $k = 3$ using the LWT-Pred assignment policy and $k = 2$ using the LSLD-Pred assignment policy. From this k the second policy shows quantitative better results than the other policy. For instance, for $k = 10$ the LWT-Pred presents an improvement of 3 times, while the LSLD-Pred shows an improvement of 10 times. The last values of K shows how requesting for 55 times more the number of requested resources for the job the dispatcher is able to make a qualitative improvement respect the orignal scenario. Thereby, for $K = 55$ the LWT-Pred improves the slowdown 26 times and the LBSLD-Pred improves the slowdown for 37 times. Note, that requesting 55 times the job requested resources does not imply that the job will use or reserve all of them. It only means that the *jScheduler* will carry out the ISIS-Algorithm to this set of resources and will select the most appropriate one. This results shows how using the second assignment policy the quality of the scheduling, based on the BSDL metric, has a qualitative improvement for $k > 4$. On the other hand, the LWT-Pred shows the similar improvements for $k > 13$.

Similar to the BSLD, the wait time improves the performance of the original scenario for $k = 2$ using the LWT-Pred assignment policy and for $k = 3$ using the LSLD-Pred. However, this metric shows different pattern that the once presented by the bounded slowdown. For lower K values ($k \in [1-8]$) the first LWT-Pred shows substantially better wait times than the LSDL-Pred. From this point the LSLD-Pred shows better wait

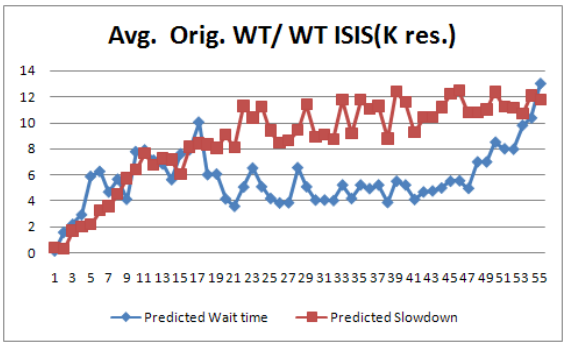

Figure 4: Improvement of BSLD against the number of requested resources

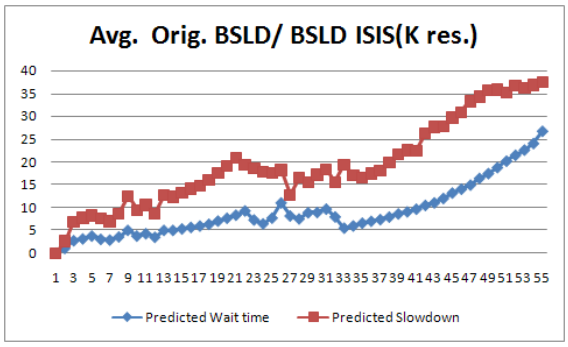

Figure 5: Improvement of Waittime against the number of requested resources

times. However, for the last k values evaluated in this work both assignment policies start converging with similar values.

The evaluations results presented in this section will be used in the next implementations of the *jScheduler* to decide which K values should be used when dispatching the submitted job. Based on the experiments, we can state that lower values are acceptable if it is required. For example when the system is really loaded, when there is an small time for carry out the scheduling (i.e: real time scheduling) or when the job request a high number of processors. In this last situation (i.e: the job request 2048 processors) the *jScheduler* should not request for $k = 55$ resources. On the other hand we have stated how this value has to be selected depending also in the metric that the user has requested to be optimized.

8 Conclusions

In this paper we have presented how the ISIS-Dispatcher techniques can be used in the XtreemOS architecture for schedule the submitted jobs (*jScheduler*). This XOS system has been designed to be deployed in large scenarios that potentially involve thousands of resources. In such systems it is not feasible to make the dispatcher contact to all the systems. It is not realistic to suppose that the dispatcher stores the information about all the resources and where they are located. Obviously, this approach would imply problems of scalability. The ADS component is the responsible to provide to the *jScheduler* the information regarding the resource that satisfy the job requirements. We evaluate the impact of the amount of resource information the dispatcher receives from the ADS during the job scheduling.

The evaluations results have shown that the *jScheduler* can achieve good performance results although not all the elements of the systems are being evaluated when carrying out the ISIS-Dispatcher algorithm. The bounded slowdown and the wait time obtained in the original system are improved by one order of magnitude using $k > 9$. Moreover, using a $k = [2,4]$ both metrics are improved up to the 50% respected the original scenarios. We have stated how the performance obtained with this approach depends on the amount of resource information available. Furthermore, we have identified that the value of k to be used should be defined depending on the amount of resources that the job requires and the metric that the user requests to optimize.

9 Acknowedgements

This paper has been supported by the Spanish Ministry of Science and Education under contract TIN200760625C0201, the Barcelona Supercomputing Center and the XtreemOS European Project under the contract FP6-033576.

References

[1] BERMAN, F., AND WOLSKI, R. Scheduling from the perspective of the application. 100–111.

[2] BERMAN, F., AND WOLSKI, R. The apples project: A status report.

[3] CHAPIN, S. J., CIRNE, W., FEITELSON, D. G., JONES, J. P., LEUTENEGGER, S. T., SCHWIEGELSHOHN, U., SMITH, W., AND TALBY, D. Benchmarks and standards for the evaluation of parallel job schedulers. *Job Scheduling Strategies for Parallel Processing vol 1659* (1999), pp. 66–89.

[4] CHIANG, S.-H., ARPACI-DUSSEAU, A. C., AND VERNON, M. K. The impact of more accurate requested runtimes on production job scheduling

performance. *8th International Workshop on Job Scheduling Strategies for Parallel Processing Vol. 2537* (2002), 103 – 127.

[5] COPPOLA, M. Design and specification of a prototype service/resource discovery system - deliverable d3.2.4, 2007.

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

[7] ERNEMANN, C., HAMSCHER, V., , AND YAHYAPOUR, R. Benefits of global grid computing for job scheduling. *5th IEEE/ACM International Workshop on Grid Computing* (2004).

[8] FEITELSON, D. D. G. Parallel workload archive, 2007.

[9] GERALD, S., RAJKUMAR, K., ARUN, R., AND PONNUSWAMY, S. Scheduling of parallel jobs in a heterogeneous multi-site environment. *JSSPP 2003* (2003).

[10] GUIM, F., AND CORBALAN, J. A job self-scheduling policy for hpc infrastructures. *Job Scheduling Strategies for Parallel Processing: 13th International Workshop, JSSPP 2007* (2007).

[11] GUIM, F., AND CORBALAN, J. Using runtime prediction with economic considerations in the isis-dispatcher. *Submitted to Transactions on Parallel and Distributed Systems* (2008).

[12] GUIM, F., CORBALAN, J., AND LABARTA, J. 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).

[13] GUIM, F., CORBALAN, J., AND LABARTA, J. Resource sharing usage aware resource selection policies for backfilling strategies. *The 2008 High Performance Computing and Simulation Conference (HPCS 2008)* (2008).

[14] HARCHOL-BALTER, M., CROVELLA, M. E., AND MURTA, C. D. On choosing a task assignment policy for a distributed server system. *Journal of Parallel and Distributed Computing 59*, 2 (1999), 204–228.

[15] HOLMES, G., DONKIN, A., , AND WITTEN, I. Weka: A machine learning workbench. *In Proc Second Australia and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia* (1994).

[16] LI, H., CHEN, J., TAO, Y., GROEP, D., , AND WOLTERS, L. Improving a local learning technique for queue wait time predictions. *Cluster and Grid computing* (2006).

[17] MORINE, P. L. C. Xtreemos project web site, 2007.

[18] PINCHAK, C., LU, P., AND GOLDENBERG, M. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. *Job Scheduling Strategies for Parallel Processing* (2002), 205–228. Lect. Notes Comput. Sci. vol. 2537.

[19] SCHROEDER, B., AND HARCHOL-BALTER, M. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *Cluster Computing 2004* (2004).

[20] SKOVIRA, J., CHAN, W., ZHOU, H., AND LIFKA, D. A. The easy - loadleveler api project. *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing Lecture Notes In Computer Science; Vol. 1162 archive* (1996), 41 – 47.

[21] SMITH, W., TAYLOR, V. E., AND FOSTER, I. T. 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* (1999), 202 – 219.

[22] SMITH, W., AND WONG, P. Resource selection using execution and queue wait time. predictions.

[23] TONI CORTES, J. C., AND PIPAN, G. Design of the architecture for application execution management in xtreemos - deliverable d3.3.2, 2007.

[24] TSAFRIR, D., ETSION, Y., AND FEITELSON, D. G. Backfilling using system-generated predictions rather than user runtime estimates. *In the IEEE TPDS* (2006).

[25] YUE, J. Global backfilling scheduling in multiclusters. *Asian Applied Computing Conference, AACC 2004* (2004), pp. 232–239.