Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Automatic Exchange of Scheduling Methods for Clusters

XtreemOS Technical Report # 2

Marko Novak[a]

Report Registration Date: July 8, 2008

Version 0.1 / Last edited by Marko Novak / July 8, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	√
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

[a]marko.novak@xlab.si

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	07/07/2008	Marko Novak	XLAB	Initial document

Abstract

Due to still increasing demand for computing power, the efficient utilization of computational clusters is becoming more and more important. During the process of utilizing such clusters, different sometimes conflicting objectives like the maximization of the overall system utilization or the minimization of the energy consumption have to be incorporated into the scheduling decision process. Therefore, the ability to dynamically adapt to changing demands becomes a highly desirable feature of cluster schedulers and job management systems. In this paper, we describe a cluster scheduling system that can automatically adapt itself to the changing system states within the cluster. Based on the different resource measurements, it can update different parameters of the scheduling policy that is currently in place or dynamically exchange the whole scheduling policy. Note that the new scheduling policy does not need to be defined at the system start. That is, the administrator can define new strategies during the runtime that are dynamically integrated. To this end, we present the main components which form our scheduling system: the Pluggable Probes and Scheduling Policies Framework and the Scheduling Configuration Changer. Furthermore, we discuss different advantages and disadvantages of our approach.

1 Introduction

The demand for compute power in science as well as in industry is permanently increasing. This is caused due to more complex applications and the introduction of new fields where computational resources are required. Furthermore, the ability to dynamically adapt to changing demands becomes more and more important especially in the business context, see e.g. Franke et. al [8]. These problems are addressed at various level, e.g. the application level by introducing Service Oriented Architectures or on the resource management level in larger data centers. The modifications and extensions at the different levels provide certain advantages and restrictions. Furthermore, the different levels are dependent on each other.

In industry as well as in scientific data centers, it becomes increasingly important to differentiate between different users, see e.g. Franke et. al [10]. That is, some customers are more important and valuable than others as the business relationship and correspondingly the prices differ. These different relationships are encoded in specific Service Level Agreements (SLA) where the usage, the usage constraints, and the data center services and costs are specified [11]. The management of the data center must incorporate all these SLAs as violated SLAs normally lead to penalties. In parallel, the management tries to minimize the needed resources. This for example affects the necessary processing nodes, the required storage, and the energy which is consumed for the processing.

In the past, data centers mainly used a central job pool where all submitted but not yet started jobs where stored. Then, a job scheduler distributes the different computational jobs onto the different nodes. In the majority of these cases, the jobs run until completion on the specified nodes. For such scenarios including individual computational clusters and computational grids, many different scheduling strategies have been developed [6, 5]. However, all these strategies did not pay attention to the different provider - customer business

relationship. Furthermore, the scheduling strategies were only used for the initial allocation. That is, the dynamic migration of computational jobs was not possible using these strategies.

In this paper, we address how the cluster operating System Kerrighed and the corresponding scheduling mechanisms overcomes most of the mentioned problems. Kerrighed is a modification of the Linux operating system and aims to manage a whole cluster as one large unit. Thus, the different user have the impression of using a single machine only. However, Kerrighed faces the same problem as more traditional cluster management systems as also here different customers have different priorities.

Here, we describe how Kerrighed has been extended to dynamically exchange scheduling strategies. Furthermore, these scheduling strategies are not only used for the initial allocation of resources to jobs but also for the dynamic migration of jobs within the whole cluster. Using these mechanisms, a dynamic adaptation to changing computational demand can be realized. Furthermore, this scheduler exchange mechanism can be used to prioritize different goals at different times. For example, it might make sense to migrate many interactive jobs that are idle during the night to a small subset of machines and switch off all other machines. This would lead to larger energy savings that directly affect the profit of the data center.

Additionally, the described scheduling strategy can adapt itself to the load of the cluster. That is, the system can automatically extend the probing period in order to reduce the overhead due to the measurements. When the load of the system is back to the normal state, the probing periods are adapted again. Thus, all decisions about scheduling strategies and about probing can be based on system parameters like the overall load, similar to the mechanisms described by Franke et. al [7, 9].

The main advantage of using Kerrighed as the base for the implementation is the reduction of the necessary overhead. All mechanisms to exchange the scheduling strategies are implemented in kernel modules. Thus, all kernel data can be accessed directly without additional overhead compared to an additional call in the normal user space.

The remainder of the paper is organized as follows. Section 2 presents job scheduling in computational clusters and dynamic scheduling of the Kerrighed operating system. Section 3 describes our cluster scheduling system, which enables the automatic exchange of scheduling methods in clusters. Section 4 discusses some of the advantages as well as some disadvantages of our approach. Section 5 concludes the paper.

2 Background

This section provides additional information on the job scheduling in computational clusters. Furthermore, the current process scheduling within the Kerrighed operating system is introduced in detail.

2.1 Job-Scheduling in Clusters

Scheduling in computational clusters is well established and data centers can use various software products for this task, e.g. Condor [14], PBSPro [12], and

Torque [18]. All these tools mainly focus on the initial allocation of jobs to resources. This resource assignment is usually performed just before the job start. The computational jobs from the different data center users are released over time. Furthermore, the jobs are normally rigid. That is the number of processors is pre-defined by the user at release date and does not vary over time. All jobs run to completion that is no preemption [16, 15] is used. Only in the case that a user specified maximum runtime is exceeded, the system stops the job execution in order to prevent the system from running faulty programs.

At the majority of installations, the scheduling system is a variation of First-Come-First-Serve (FCFS) [19], or EASY backfilling [13]. These scheduling strategies are static and do not vary over time. Newer system management approaches incorporate many different scheduling objectives during the decision process, see e.g. Franke et. al [10]. This leads to a dynamic exchange of scheduling algorithms depending on the preferences of the data center provider and the different systems states. To this end, a system state classification is learned offline and later applied online. This is a restriction as the later readjustment of the system classification or the set of possible scheduling algorithms is in the current implementations only possible by stopping the whole system for a short period of time. Within this work, we present a method to overcome these problems.

2.2 Process Scheduling in Kerrighed

Kerrighed [3] is a community project that was initiated as a research project to provide simple operating systems for clusters [17]. Its development is now supported by a dedicated company called Kerlabs [2].

Process scheduling in Kerrighed differs from traditional approaches on clusters thanks to its Single System Image (SSI) property. The SSI property is the ability of a system to hide the heterogeneous and distributed nature of the available resources and present them to users and applications as a single unified computing resource. In other words, an SSI cluster of 10 uniprocessor computers is seen by the user as a single 10-processor computer. Kerrighed is fully decentralized (there is no front-end node) and is implemented as an extension of the Linux kernel that adds SSI features like:

- **Cluster-wide process management:** processes get cluster-wide PID's, the **fork()** system call can create processes on remote nodes, signals are sent to processes whatever their location, etc.;
- **Cluster-wide shared memory:** a parallel application using the shared memory communication paradigm can be transparently distributed over the cluster nodes.

In particular, thanks to the SSI property, Kerrighed can transparently create processes on remote nodes (*remote fork*) when a program calls the **fork()** system call, and transparently migrate processes during their execution.

For cluster-wide process scheduling, Kerrighed provides a customizable global process scheduler[20]. This service is integrated in a framework allowing administrators to design customer specific scheduling policies using the low-level process management features like remote fork and migration. The work presented in this paper is not based on the framework presented in [20], but on a new

implementation featuring improved flexibility and functionalities which are described in Section 3.2. The main concepts driving both versions of Kerrighed's global scheduler framework are:

- **Separate global scheduling from local scheduling:** global scheduling policies only take care of choosing which node executes which process, and let the Linux scheduler of each node schedule processes on the node's CPUs.
- **Instantiate global schedulers on all nodes:** global schedulers are fully distributed. This property is required to be able to dynamically add or remove nodes to the cluster without having to stop it.
- **Build schedulers out of building blocks:** to help the design of custom schedulers, the framework allows to connect scheduling components together and provides an API to let components publish/request data and notify events. Scheduler designers can then reuse these components for different schedulers.
- **Implement at kernel-level:** for performance reasons all the framework and all scheduling building blocks are implemented at kernel-level. The main performance issues are the overhead to monitor system properties frequently (doing this in userspace adds the overhead of at least one system call at each read, and many process-related states are not exposed to userspace), and efficient parsing of processes, for instance find a good candidate to migrate in order to balance the CPU load.
- **Separate data/event collecting from process placement/migration strategies:** the main types of components defined by the framework are *probes* and *policies* (called global scheduling managers [20]). Probes collect data or notify events from kernel internals, and policies implement the process placement/migration strategies using information provided by the probes.
- **Allow to modify schedulers at run-time:** scheduling components can be loaded and linked together at run-time, and unlinked and unloaded at run-time as well. This allows administrators to change of scheduler without stopping the cluster, for instance to upgrade the policy, or to adapt to a workload change.

Although this framework provides administrators with a large flexibility in defining scheduling policies adapted to their use case, this framework does not solve all the issues exposed in Section 2.1. In particular, Kerrighed does not relieve administrators from having to change scheduling policies each time the workload characteristics change. Using Kerrighed's ability to change global scheduling policies at run-time, we present how this can be solved in Section 3.

3 Our Approach

Our main goal was to design a scheduling system that would be able to adapt itself automatically to various state changes in the cluster (e.g. increased load

in the cluster, load imbalance among the nodes, memory shortage on particular nodes, increased network traffic, etc.). No human intervention should be needed. This would significantly improve the adaptability of the clusters and thus increase its utilization.

For the implementation, we haven chosen the Kerrighed system, however our approach can be used on every operating system that enables process migration among various cluster nodes (we need a process migration in order to perform dynamic scheduling of the jobs that are already executing).

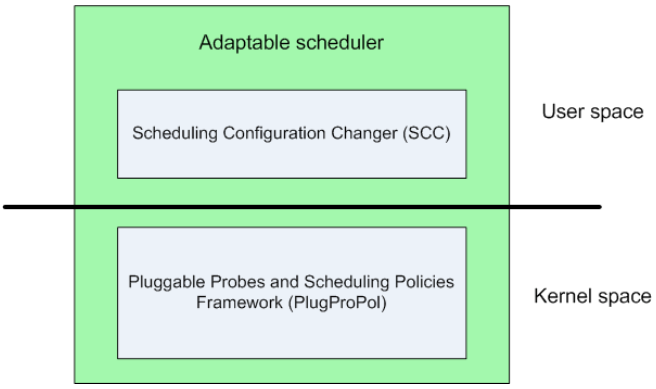

Figure 1: Adaptable scheduler architecture.

In order to separate the logic that triggers the changing of scheduling algorithms from the infrastructure that performs the actual changing, we have decided to separate our system into two modules (see Figure 1):

- Pluggable Probes and Scheduling Policies framework
- Scheduling Configuration Changer

The modularity of the framework is a key feature to simplify the implementation of the probes and the scheduling policies, as well as all the necessary logic for triggering the changes.

3.1 Terminology

Before we start describing our approach regarding the dynamic exchange of scheduling methods, we define the basic notion which are used throughout the whole description.

Probe: an entity for measuring different resource properties (e.g. CPU load, CPU speed, total memory, free memory). A probe collects information (data or events) and makes them available to other scheduler components. Scheduler designers can implement these probes as separate Linux kernel modules and insert them dynamically into the kernel. Probes may collect data already computed by the kernel (CPU average load for instance) or compute other data (alternative notion of CPU load, like the one of Mosix [4] for instance), thus extending the set of resource properties that are being measured.

Scheduling policy: an implementation of a job scheduling algorithm. In Kerrighed, a scheduling policy is in charge of selecting a proper node for a particular process. Scheduling policies base their decisions on data/events collected

by probes. An example of a scheduling policy is a migration-based, sender-initiated load balancing policy. This policy takes resource properties from one or more probes as input and when it detects that the local load is higher than the loads of remote nodes, it tries to migrate processes to balance the load.

Filter: an intermediate entity taking data/events as input and producing a filtered output. Filters are useful to share probes between several scheduling policies and adapt the probes' outputs to each scheduling policy, or to implement modular features of a scheduler. Filters can be chained to implement complex filters out of simple ones. Filters implement for instance different caching policies of data collected from remote nodes, or block events unless the value of the event source exceeds a threshold.

Scheduling configuration: a set of probes, scheduling policies and filters that is loaded on a cluster node at a particular moment. Only one scheduling configuration can be selected at a given moment. Different scheduling configurations correspond to different cluster states (the cluster states are defined using various resource measurements). For example, we can define two separate scheduling configurations: one is selected when the cluster load is low (e.g. the average CPU usage is smaller than 50%) and the other is selected when the load is high (e.g. the average CPU usage is greater than 50%).

Selector: an entity which is in charge of selecting user-defined scheduling configuration based on different resource properties. The selector implements all the necessary logic for deciding which scheduling configuration should be chosen under given conditions. In order to make selectors as configurable as possible, they have to be implemented as a separate plug-ins and added to the scheduling system.

3.2 Pluggable Probes and Scheduling Policies Framework (PlugProPol)

PlugProPol is an infrastructure which enables scheduler designers to write their own probes and scheduling policies and add them to the system at runtime (without the need to restart the cluster). This infrastructure is the basis for the Scheduler Configuration Changer presented in Section 3.3. First we describe the design goals of the PlugProPol framework, and second we give an overview of its design and implementation.

3.2.1 Features

The main goal of Kerrighed's scheduler framework is to provide administrators with highly configurable scheduler configurations. For instance, if the administrator has to setup a new scheduler based on disk usage, he only implements a proper probe and plugs it to the framework runtime. After that, he can start collecting the data about the disk usage immediately. All these operations can be done without recompiling the operating system kernel, and without even rebooting the cluster. The administrator should also be able to change the set of locally measured resources very easily and thus adapt it to his current needs, local system load, etc.

This goal induces three main modularity needs:

- Allow administrators to build schedulers with building blocks.

- Separate a scheduler architecture into basic functionalities that can be implemented by building blocks. The basic functionalities are probes, scheduling policies, and filters.

- Allow administrators to dynamically configure, connect, and disconnect building blocks at run time.

For instance, an administrator should be able to build a process migration-based CPU load balancer out of two building blocks: a CPU probe measuring periodically the CPU loads of the nodes, and a generic migration-based load balancing scheduling policy. To do this, the administrator uses the framework to configure both components, and simply connect them together. The administrator is then able, at runtime, to refine the scheduler, for instance introducing a filter relaying newly computed local load to the scheduling policy only if it exceeds a threshold above which the performance of local processes may decrease, or changing the period of CPU load computation to decrease the operating system overhead. Figure 2 shows an example of such a refined CPU load balancer, where the scheduling policy selects candidate processes according to the load they induce. The scheduler was added a threshold filter as just described in order to decrease the overhead of process scanning and comparisons between the local load and remote loads. Moreover, the scheduler was added a caching filter that decreases the overhead of retrieving the loads of remote nodes. Those last two building blocks could be added at anytime during the execution, and can be reconfigured, exchanged, or removed at anytime.

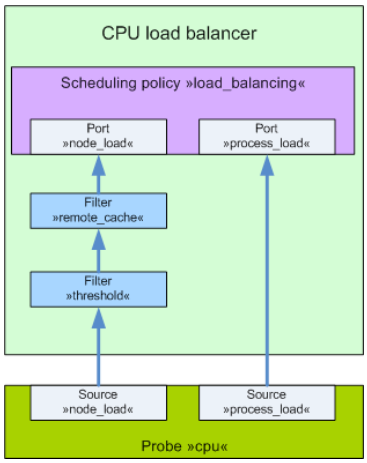

Figure 2: Modularity of schedulers with probes, scheduling policies, and filters.

To the three main needs just mentioned, for performance reasons, we added the constraint to implement the framework and all the scheduler components at kernel-level, the scheduler components being implemented as loadable modules for the Kerrighed kernel. As we already mentioned in Section 2.2, the main performance issues are the overhead to frequently monitor system properties from userspace when they are computed as kernel-level state, provided that an interface exists, and efficient parsing of processes, for instance find a good candidate to migrate in order to balance the CPU load.

3.2.2 Design and implementation

The PlugProPol scheduler framework is very similar to the former Kerrighed scheduler framework [21]. Besides the implementation details, the main differences are the following:

- the scheduler configuration is based on the configfs virtual file system, instead of XML files [21];
- PlugProPol introduces filters, which are a generalization of local analyzers [21];
- PlugProPol introduces *process sets* (not described in this paper though), which allow administrators to define several schedulers and let each of them act only on configured sets of processes.

The configfs [1] virtual file system provides a file-system based user interface to create, configure, and destroy kernel objects. The root of the configfs file system provides one directory for each registered subsystem. Each of these directories and sub-directories represent kernel objects. All file entries in a directory represent object attributes. To create an object, the user simply calls the **mkdir** system call, for instance using the `mkdir` shell command. Similarly, to get or set the value of an object's attribute, the user simply calls the **read** and **write** system calls, for instance using the `cat` and `echo` shell commands. Kernel objects can be created under other kernel objects as sub-directories, or linked together through symbolic links from one object directory to a file entry in another object's directory. Finally, kernel objects are destroyed by simply removing their representing directories.

From this properties of configfs, PlugProPol implements all its needed dynamic configurability interfaces using the configfs interface and the automatic module loading feature of the Linux kernel. Components are represented as kernel objects (directories in configfs), and are linked either by creating a component under the directory of another one (see the description of filters below), or by drawing a symbolic link from one component directory to an entry in another component directory (see the description of probes below). The name of the kernel module implementing a component is derived from the directory name of the component when it is created, which enables the PlugProPol framework to automatically load the needed kernel modules on demand.

The PlugProPol implementation in Kerrighed also provides an SSI view of the scheduler configuration: object creation, destruction, or attribute settings are automatically replicated on all nodes by the framework. However, to also apply to non-SSI cluster operating systems, the work presented in this paper is based on a modified version of the PlugProPol framework, in which the SSI view of scheduler configuration is disabled.

PlugProPol (Figure 3) defines five component types: probes, scheduling policies, filters, schedulers (not described in this paper), and process sets (not described in this paper).

Probes, scheduling policies, and filters are implemented as Linux kernel modules that are dynamically loaded when the first component that they implement is instantiated, and unloaded when no component that they implement exist anymore in the configured schedulers. Schedulers and process sets are generic components implemented by the framework.

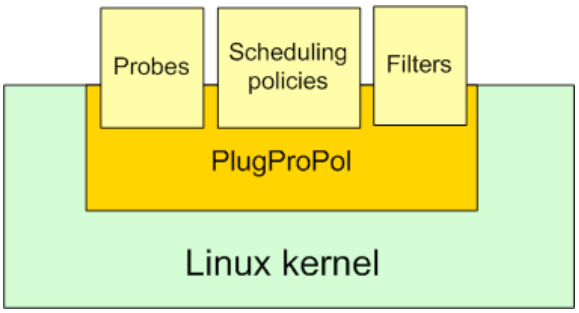

Figure 3: Architecture of the PlugProPol framework.

In the PlugProPol framework, the interfaces for data/event flows are derive from *sources* (implemented by probe sources and filters) and *sinks* (implemented by filters and by scheduling policies ports) concept. Sources produce data and/or events, that are retrieved by sinks. Sources can publish events (for instance to notify new values of their data) for which sinks define notify methods called **update_value**, and sinks can collect data from sources using the **get_value** methods of sources. To protect against dynamic removals of components, a component never directly refers to a connected component but instead always calls functions of the framework that will in turn call methods of the connected component with the needed precautions.

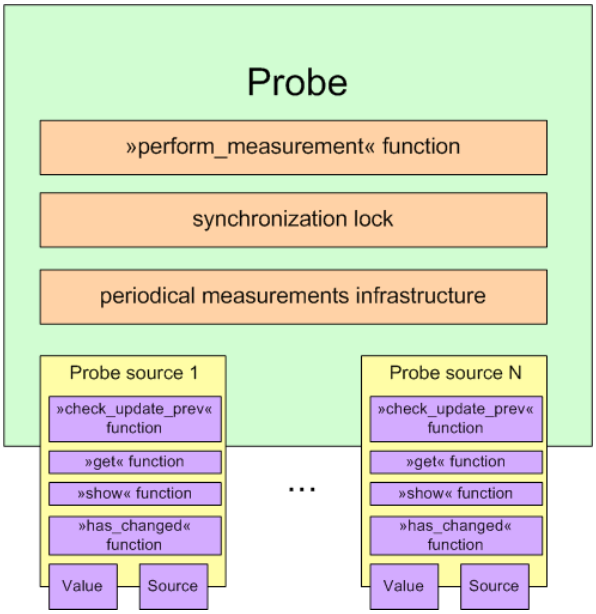

Figure 4: Architecture of probes.

Figure 4 shows the architecture of probes. Every probe contains a set of data/event sources defined by the designer, and represented as sub-directories of the probe in configfs. Probe sources are linked to sinks (filters or scheduling policies ports) by drawing symbolic links from their representing directory to

arbitrary-named entries in the sinks' directories. Probes doing periodic measures define a **perform_measurement** method that will be called periodically by the framework using the period configured for the probe.

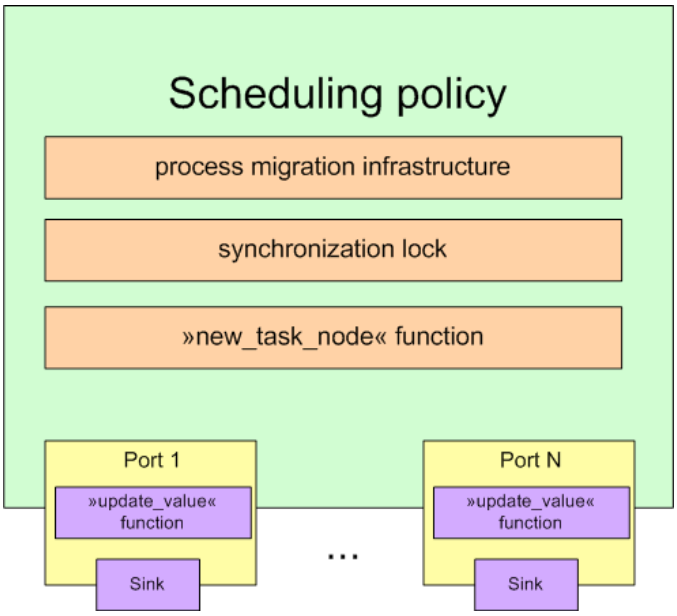

Figure 5: Architecture of scheduling policies.

Scheduling policies (Figure 5) provide a **new_task_node** method in order to choose the target node when the **fork** system call is called. They collect their information using ports, that are sub-directories implementing sinks.

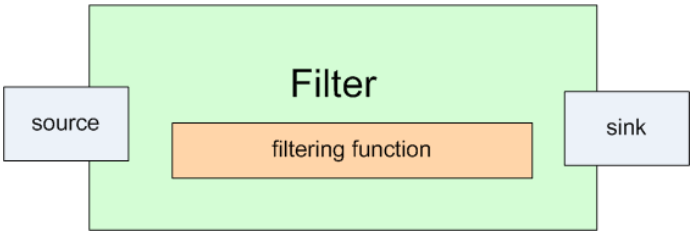

Figure 6: Architecture of filters.

A filter (Figure 6) is represented by a directory, and implements a source, connected to a scheduling policy port or to another filter sink, and a sink that can be connected either to a filter source or to a probe source. To connect a port (resp. filter sink) to a filter source, the administrator creates a sub-directory in the port's (resp. filter sink's) directory, which creates a new filter and connects it to the port (resp. filter sink).

3.3 Scheduling Configuration Changer

The Scheduling Configuration Changer (SCC) is the component that is in charge of choosing the proper scheduling configuration (see the definition of the scheduling configuration in Section 3.1) based on different resource measurements (e.g. CPU load, ...). By doing that, it adapts the scheduler to the changing state of the cluster. The selection of a scheduling configuration is completely automated. The user only creates a proper settings file in which he defines all the scheduling configurations. He also implements the selector that is used for selecting a proper scheduling configuration (see the definition of selector in Section 3.1). After that, he loads the settings file to the SCC. The SCC then takes care of loading, unloading and parameter changing of all the PlugProPol components (i.e. probes, policies, filters). It also takes care of connecting the components. Since no user intervention is needed, the scheduler is much more adaptable to the changes in the state of the cluster.

The SCC runs in the user-space. This way, as we already mentioned in Section 3.2.2, the SCC implementation can be much more sophisticated. This allows us to provide, among other things:

- a more robust and user-friendly interface: since SCC runs in user-space, we can implement settings files as XML files. Users can read and modify such files very easily. If the SCC would have been implemented in kernel-space, just using settings files, it would haven been very complicated to just parse XML settings files,

- a better working scheduling configuration selectors: in user-space, we can implement much more complicated logic for selecting scheduling configurations. For example, we could use some computational intelligence libraries for deciding which configuration to choose, see for example Franke et. al [9]. In kernel-space, it would have been impossible to use any user-space libraries. As a consequence, the selection logic would have been much simpler and thus less efficient.

The disadvantage of running SCC in the user space is the overhead induced due to the system calls which are needed for SCC to communicate with PlugProPol which is executing in kernel space. We claim this overhead to be insignificant, since the communication with the PlugProPol is not frequent. It occurs only once every few seconds.

The SCC component consists of two main parts (see Figure 7). The first part is the **settings file parser**. It is used for parsing scheduler's settings files and extracting all the necessary data about scheduling configurations. We decided to write scheduler settings in XML format. This format can be read and modified very easily by the users and can easily be parsed by computers. The structure of the settings file is presented in the Figure 8. This file contains the definitions of all the available scheduling configurations. Each scheduling configuration contains a list of actions a PlugProPol framework has to perform when a given scheduling configuration is selected. The most important actions were already mentioned in previous sections of the paper: loading and unloading of probes/policies/filters, setting their parameters (e.g. for probes, we can set their probing frequency, for scheduling policies, we can set the thresholds which trigger process migrations), connecting scheduling policies to probes, etc. The

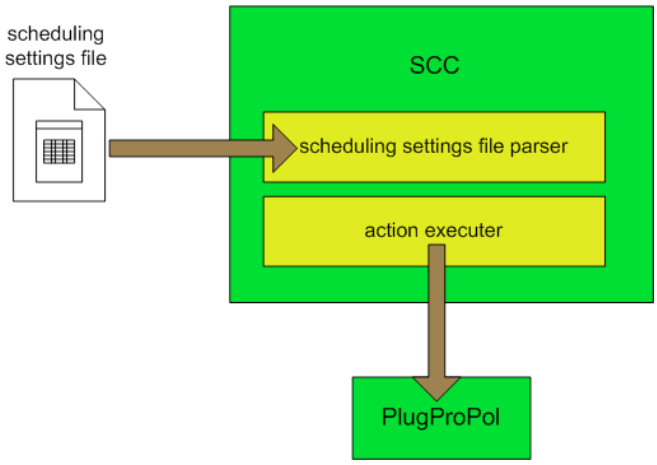

Figure 7: Architecture of the Scheduling Configuration Changer.

settings file also contains a path to the selector, a component which initiates the exchange of scheduling configurations. The selector is implemented by the user and contains all the necessary logic for selecting the proper scheduling configuration based on different resource measurements.

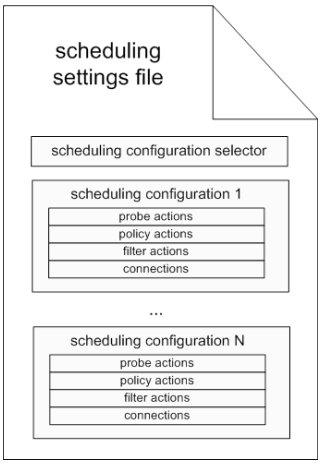

Figure 8: Structure of the scheduling settings file.

The second part is the **action executer**. It is in charge of initiating commands to the PlugProPol system to load and unload probes/policies/filters, change their parameters, make connections, etc. These actions are performed every time the scheduling configuration is replaced. The component also takes care of reading resource measurements from the probes and passes them to the selector. Obviously, the action executer component is tightly coupled with the PlugProPol system. Basically, all it does is execute configfs commands on PlugProPol (as mentioned in Section 3.2.2, this is how the PlugProPol actions are triggered).

The single SCC instance is only in charge of a node it is running on. As a

consequence, each cluster node needs to host its own SCC instance. We have chosen this per-node approach since it allows us to provide much more efficient scheduling strategies than the approach with a single centralized SCC instance. At a given point in time, the scheduling configurations that are selected on each of the nodes do not have to be the same. Each node can adapt the scheduling configuration to its local load. Obviously, such scheduling is much more adaptable than the one where a single global scheduling configuration would be chosen based on a average global state in the cluster. For example, consider an SCC system that is configured in a way that adapts, among other things, the probes' measurement frequency to the CPU usage: when the CPU usage is high, the probe measurements are less frequent compared to when the CPU usage is low. This way, the SCC system reduces the probing overhead when more processing power is needed. It is obvious that having a separate SCC service on each cluster node which only takes care of changing the measurement frequency of that single node is much more adaptable than having a global SCC service and a common probing frequency for the whole cluster. Of course, the per-node scheduling also has disadvantages. Since we are dealing with multiple SCC services we have to provide a way for them to exchange data. This makes the SCC system more complex. Furthermore, there is a possibility that a group of nodes will continuously swap the processes among themselves due to different scheduling configurations that are selected on each of the nodes. The implementation of mechanisms that would prevent that is a part of our future work.

In order for a user to establish a customizable scheduler in the cluster, he has to create a proper settings file in which he describes all the scheduling configurations for different states of the cluster. Each scheduling configuration needs to contain the list of probes, policies and filters that need to be loaded when the configuration is selected. Additionally, it has to include the list of all the connections between the PlugProPol entities. The user also needs to provide the implementations of all the necessary PlugProPol entities. He can use existing entities or implement them by himself.

One of the most important things when establishing scheduling is implementing selector. As already mentioned in Section 3.1, the selector implements all the necessary logic for selecting a particular scheduling configuration based on different resource measurements. Since the selector has to be highly customizable (it has to be able to handle diverse user-defined cluster states), its logic can get very complex. As such, they cannot be realized with a set of predefined rule types (such as if-then rules). Instead, they have to be implemented as separate plug-ins and added to SCC via settings files.

4 Discussion

The system we presented in Section 3 has many advantages over traditional job management systems. Since it allows process migration to different cluster nodes while the jobs are executing, it enables much more optimized distribution of the load in the cluster. At any time, the processes from the overloaded nodes can be moved to the less loaded nodes. This greatly improves the utilization of the cluster and makes the average response time for the jobs shorter.

The second advantage is the ability to automatically exchange scheduling

configurations based on the state of the cluster (as mentioned earlier, the cluster states are defined using various resource measurements). No user intervention is required. This is beneficial since cluster administrators are releaved of the burden of having to periodically check for cluster load and manually adapt scheduling algorithms to it. Moreover, automatized exchange of scheduling configurations enables the scheduler to adapt to the changing cluster state much more quickly than with the manual reconfiguration. The cluster benefits from a better adaptability in various ways. For example, automatic exchange of scheduling configurations improves utilization of the cluster even further, it also enables lower energy consumption (see Section 1), which is becoming more and more important these days.

On the other hand, since the system is in an early development stage, it also has some drawbacks. For example, the SCC framework does not offer much functionality that would ease the implementation of selectors for the user (e.g. some set of predefined components that would include simple rules for exchanging scheduling configurations. These rules would then be combined together to form more complicated rules. This concept is the same as with Lego bricks). Since there are no predefined components available, the whole burden of implementing logic for the selector lies on the user. Furthermore, since at a given point, different nodes can have different scheduling configurations with different goals selected (i.e. multiple scheduling configurations exist in the cluster at a given moment), this could lead to process swapping (i.e. the set of processes is continuously migrated within a group of nodes). For example, if a user is not careful and allows situations in which some nodes use a scheduling configuration for optimizing CPU usage and the others use a configuration for memory optimization, it could happen that under certain conditions, both groups would start swapping processes. Obviously, the swapping introduces unnecessary overhead and significantly deteriorates the cluster performance. So far, no mechanisms for handling such issues have been examined. We are planning to do that as a part of our future work.

5 Conclusion

In this paper, we described a cluster scheduling system that can automatically adapt itself to the changing system state of a cluster. Since no human intervention is needed, the system significantly improves the adaptability of the cluster and increase its utilization. For the implementation, we have chosen the Kerrighed operating system, however our approach can be used on every operating system that enables process migration among various cluster nodes. We described both components of our scheduling system: the Pluggable Probes and Scheduling Policies (PlugProPol), a framework for managing probes, policies and filters, and Scheduling Configuration Changer (SCC), an implementation of infrastructure for selecting scheduling configuration based on resource measurements. We believe our system has many advantages over traditional scheduling systems. For example, since no user intervention is required, the cluster administrators are releaved of the burden of having to periodically check for cluster load and manually adapt scheduling algorithms to it. Also, due to the automated exchange of scheduling configurations, the scheduler is able to adapt to the changing cluster state much more quickly than with manual recon-

figuration. This improves the utilization of the cluster. Of course, our system also has some drawbacks. In this stage, the SCC system does not offer much functionality that would ease the implementation of scheduling configuration selectors, so the whole burden of implementing logic for selecting scheduling configurations lies on the user. The second disadvantage is the possibility of process swapping between a subset of nodes in the cluster. To overcome these problems a somewhat deeper understanding of the problem area is necessary. Thus, we decided to put the solving of the issues above as a part of our future work.

6 Acknowledgements

The work has been partially funded by the European Community under the FP6 IST project XtreemOS, http://www.xtreemos.eu

References

[1] configfs pseudo file system. `http://lwn.net/Articles/148973/`, `http://lwn.net/Articles/130342/`, `http://lwn.net/Articles/148987/`, 2008.

[2] Kerlabs. `http://www.kerlabs.com/`, 2008.

[3] Kerrighed. `http://www.kerriged.org/`, 2008.

[4] A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[5] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore*, volume 2536 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2002.

[6] C. Ernemann and R. Yahyapour. *"Grid Resource Management - State of the Art and Future Trends"*, chapter "Applying Economic Scheduling Methods to Grid Environments", pages 491–506. Kluwer Academic Publishers, 2003.

[7] C. Franke, F. Hoffmann, J. Lepping, and U. Schwiegelshohn. Development of scheduling strategies with genetic fuzzy systems. *Applied Soft Computing Journal*, 8(1):706–721, January 2008.

[8] C. Franke, A. Hohl, P. Robinson, and B. Scheuermann. On business grid demands and approaches. In *Proceedings of the 4th International Workshop on Grid Economics and Business Models (GECON 2007)*, volume 4685 of *Lecture Notes in Computer Science*, pages 124–135. Springer, 2007.

[9] C. Franke, J. Lepping, and U. Schwiegelshohn. Genetic fuzzy systems applied to online job scheduling. In *Proceedings of the 2007 IEEE International Conference on Fuzzy Systems*, pages 1573–1578, London, June 2007. IEEE, IEEE Press.

[10] C. Franke, J. Lepping, and U. Schwiegelshohn. On Advantages of Scheduling Using Genetic Fuzzy Systems. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Proceedings of the 12th Job Scheduling Strategies for Parallel Processing*, volume 4376 of *Lecture Notes in Computer Science (LNCS)*, pages 68–93. Springer, January 2007.

[11] Ca. Franke and P. Robinson. Autonomic provisioning of hosted applications with level of isolation terms. In *Proceedings of the Fifth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASE08)*, pages 107–112, Belfast, UK, 2008. IEEE Computer Society. to appear.

[12] PBSPro homepage. http://www.pbsgridworks.com, March 2007.

[13] D. A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP95)*, volume 949 of *Lecture Notes in Computer Science (LNCS)*, pages 295–303. Springer, 1995.

[14] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[15] D. McLaughlin, S. Sardesai, and P. Dasgupta. Preemptive scheduling for distributed systems. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems*, September 1998.

[16] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, October 1959.

[17] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an efficient single system image cluster operating system. *Future Generation Computer Systems*, 20(2), January 2004.

[18] TORQUE Resource Manager. Overview. http://old.clusterresources.com/products/torque, March 2007.

[19] J. J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu. Scheduling parallelizable tasks to minimize average response times. In *Proceedings of the 6th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA94)*, pages 200–209. ACM Press, New York, USA, June 1994.

[20] G. Vallée, R. Lottiaux, L. Rilling, J.-Y. Berthou, I. Dutka-Malhen, and C. Morin. A case for single system image cluster operating systems: the kerrighed approach. *Parallel Processing Letters*, 13(2), June 2003.

[21] G. Vallée, C. Morin, J.-Y. Berthou, and L. Rilling. A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 91, Washington, DC, USA, 2003. IEEE Computer Society.