A Tool for Isolating Performance in General-Purpose Operating Systems

Valéria Q. Reis*
Computer Science Department – PUC-Rio
Rio de Janeiro, Brazil
vreis@inf.puc-rio.br

Renato F. G. Cerqueira
Computer Science Department – PUC-Rio
Rio de Janeiro, Brazil
rcerq@inf.puc-rio.br

ABSTRACT

General-purpose Operating Systems do not provide effective mechanisms for application processing reservation. For this reason, some initiatives aim at guaranteeing processing by instrumenting kernels or by isolating the performance through the creation of virtual machines. As will be described in the present paper, CPUReserve works differently from these approaches. It is a processing reservation system that runs at user level. Because CPUReserve presents a client-server architecture and significant scalability – as suggested by the experiments carried out – it can be used in distributed and shared environments just like computational grids.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling*

General Terms

Management

Keywords

Scheduling policies, Distributed computing, Processing reservation, User-Level scheduler

1. INTRODUCTION

General-purpose Operating Systems do not treat application classes in any specific manner, and thus are inefficient in managing the Quality of Service of some applications. When scheduling is performed by time- or space-sharing algorithms, different priority levels among the applications are not distinguished, aside from the fact that these algorithms are very conservative – the time or space slices allotted cannot be transferred to other processes even when they are not being used. On the other hand, when priority-based process-scheduling systems are used, applications classified as having low priority might have to wait too long.

Due to their own nature, shared computational environments pose certain challenges to resource management. Ensuring that users do not overload machines or violate usage policies, as well as ensuring a minimum service standard for all users, can be very difficult tasks when there are no tools to limit the use of system resources. Implementing resource-reservation mechanisms can facilitate management by setting lower and upper limits for processing, memory, disk or network usage. Particularly, the present study focuses on processing because many of the applications submitted to shared environments are CPU-bound.

In the case of computational grids, defining processing reservation can prevent overload of system nodes while giving machine providers the right to set the processing capacity limits they wish to offer. In the case of on-demand computing, reservation can guarantee a minimum service standard for users. The same applies to multimedia applications, which require a minimum and periodic processing.

Some initiatives, such as Resource Kernels or Resource Containers, seek to develop processing-reservation mechanisms by creating kernel extensions, with the purpose of ensuring timely access to the resources of an Operating System [9, 7, 1]. However, there is a new tendency to use virtual machines for this purpose. Virtual machines offer users a customized environment in which computer resources are exclusively dedicated to the processes required by the current user. This isolation of the environment results in greater control over the resources used and in increased security, because a process in a given virtual machine cannot access data in another virtual machine [6, 11].

Taking a different approach from Resource Kernels and virtual machines, some initiatives have attempted to manage resource reservation at user level, thus avoiding the need for kernel recompiling or system overload by instancing a large number of virtual machines. An example of such solution is DSRT – Dynamic Soft Real Time CPU Scheduler [3]. Created in the late $1990's$ with the purpose of treating processing reservation for multimedia applications, DSRT was used in several projects. However, it did not evolve enough to be applied in more modern scenarios, such as opportunistic grids, utility-computing environments, and multi-processed architectures. These limitations combined to the difficulty of adding new resource-sharing policies in this system have motivated the development of a new reservation manager, called CPUReserve, which is the objective of study of this

*Sponsored by a CNPq doctoral fellowship.

MGC '08, December 1-5, 2008 Leuven, Belgium.

paper.

The rest of the paper is organized as follows: in Section 2, other works related to CPUReserve are reviewed. In Section 3, implementation details of the system and its architecture are presented. In Section 4, the experimental evaluation and limitations of CPUReserve are described. Section 5 outlines a proposal to integrate user-level processing-reservation approaches, also including virtual machines. Finally, Section 6 concludes this paper.

2. RELATED WORK

Processing reservation is often related to Resource Kernels. A resource kernel is a kernel that has been modified to manage resources by means of reservation models. It allows ensuring access to a certain portion of a machine's resources while running an application, using functionalities provided by the core of the OS. RT-Mach and Linux/RK are two of the main Resource Kernels referred to in the literature [9, 7]. Both of them, as well as CPUReserve, provide ways for specifying the applications' execution periods. However, unlike CPUReserve, RT-Mach kernel and Linux/RK need kernel recompiling for the changes in their code to make effective. Since 2.6.23 kernel release, the Linux Kernel Organization[1] has adopted the Completely Fair Scheduling (CFS) as its new task scheduler. CFS proportionally divides CPU among the system's processes according to the their corresponding weights. However, CFS does not provide an admission control for new applications, nor does it ensure applications QoS guarantees that require a specific amount of CPU allocation at a constant periodicity as CPUReserve does.

Recent initiatives have been using virtual machines as a means to ensure processing reservation, since virtual machines create closed execution environments that provide security and isolation for application performance [6, 11]. Xen is one of the most relevant monitors in the literature [2]. In Xen, the monitor can load different schedulers during the host operating system booting phase, including the sEDF scheduler, which enforces domains to execute during a time slice within a period. Although processing reservations can be assigned to domains, there is no mechanism for isolating the performance of the processes running inside each domain. The only way to achieve performance isolation using virtual machines is to execute one domain for each application. This practice, however, may generate a great overhead to create and manage big numbers of virtual machine instances. On the other hand, CPUReserve can isolate part of a machine processing capacity and assign it to a specific process with an almost insignificant resource usage.

Like CPUReserve, DSRT manages processing reservation at user level [3]. In DSRT as well as in CPUReserve, processing reservations are implemented by temporizers which, upon expiring, handle the priorities of client applications on the current Operating System. However, on the former system, client and server applications must run in the same machine, as communication between these entities is made by memory sharing. This implementation makes the use of DSRT in distributed environments difficult. In CPUReserve, the client-server communication is made through sockets.

DSRT was designed to respond to the needs of multimedia applications, so it allows classifying applications according to the processing profile, which may adopt different periodicity patterns. It also provides a reservation API that can be added to application code, to allow a more precise control over the resources. On the other hand, CPUReserve was designed to respond to the needs of computing intensive applications in scenarios such as the ones found in opportunistic grids and multi-processed architectures. Consequently, CPUReserve can take advantage of idle processing in machines in addition to provide mechanisms for processor reservation in machines with more than one processor.

Finally, CPUReserve was designed seeking to prioritize code modularization, so that the reservation-implementation mechanism is independent from the reservation policies. This separation between reservation mechanisms and reservation policies allows introducing new prioritization parameters in CPUReserve.

3. CPURESERVE

The disadvantages identified in processing reservation performed by Resource Kernels and by virtual machines have motivated the implementation of CPUReserve, a user-level processing reservation system. Similarly to DSRT, CPUReserve manages processes through system calls that dynamically change priorities of processes, causing them to consume more or less processing in a given time span.

CPUReserve uses the client-server model with communication among processes made through sockets. To run the server, the process needs to be launched at a given port, defining a bitmask that informs in which processors in the machine the reservations are to be managed and the system's processing limit[2]:

```
./server <port> <processors_decimal_set> <system_limit>
```

To run the client, one needs to run a command such as

```
./client <mach:port><period><slice><cons><exec><params...>
```

where `mach:port` refers to the machine name combined with the TCP port number where the server is listening, and `slice` is the processing time percentage required by the application at each period specified in the server configuration. Field `cons` (*work-conserving*), if set to 0, restricts the amount of CPU allotted to the process specified in the message. If this field is 1, the amount of processing per period can be larger if the running machine has idle processing. Fields `exec` and `params`, together, correspond to the command line of the application to be executed.

If the client's request is accepted, the application will be allotted the CPU percentage specified in the request. Reservation parameters can be changed by means of a command such as

```
./adapt <mach:port> <period> <slice> <cons> <pid>
```

where `pid` is the identifier of the process being executed by the server.

To prevent starvation of the processes running outside the CPUReserve server, including the server itself, a reservation limit is set using variable RESERVATION_LIMIT, which is 0.8*num_reserved_processors by default[3].

Reservation and adaptation activities cover the whole process tree. New processes share the time slice reserved for the

[1] http://www.kernel.org/

[2] Typically, this value is set to 1, meaning, the system is not limited by another entity such as a hypervisor, for example.

[3] This value was determined through the experiments described in Section 4.

process that created them. This method prevents children processes from bypassing the agreement regarding processing usage.

3.1 Architecture

The CPUReserve server is composed of two main threads: one to monitor CPU usage (idle CPU and CPU time used by each process), and one to await client connections. The connection-waiting thread listens, on the port set when the server started running, for reservation or adaptation requests.

Figure 1 illustrates the server architecture and the interaction among its components. Upon receiving a message from the client through the IO-treatment thread, the server starts another thread to treat this request. This latter thread treats reservation and adaptation requests differently, but in any case it checks the message's parameters for consistency and assesses whether the reservation requested can be provided (admission control). In the case of reservation, if the client request can be serviced, the specified process is created, and its processing consumption is monitored. New children of this process will also be monitored to ensure that they will share the time slice defined for the parent process. In the case of the adaptation request, new process reservation parameters are updated and new alarms are configured.

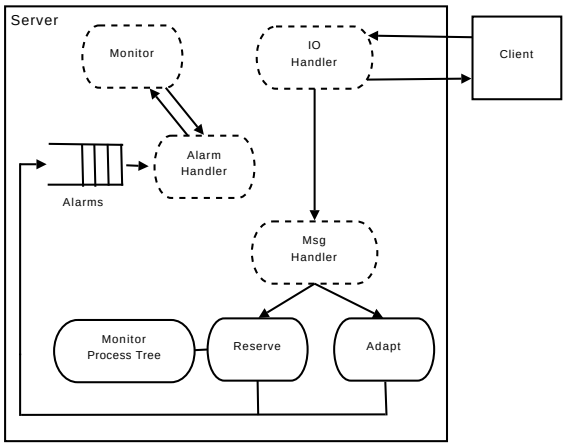

Figure 1: Architecture of the reservation server.

Processing-reservation control is made through alarms that expire after a processing time slice. The alarms are ordered, with those that expire earlier being first in line. When an alarm expires, a thread is created to treat it. This thread checks the processing time of the process that generated the alarm and makes decisions regarding continuing or interrupting the execution of that process. The monitoring and alarm-treating threads exchange information. Such exchanges occur when the alarm-treating thread wants to know whether the machine has any idle processing capacity, in order to allocate it to a work-conserving type of process.

3.2 Implementation Details

When the server is initiated, its execution is bound to a given set of processors. This binding restricts not only the server's execution but also that of the processes it manages. Their execution is ensured by calls to the Operating System that makes the processes bindings. CPUReserve is implemented using the Linux API for its Linux version and win32 API for its Windows version. In both implementations, it is assumed that the operating system scheduler is priority concerned and that processes in a same priority queue divide the processing capacity equally, in a round-robin fashion, for example.

To manage the scheduling of the processes requested by the clients, the server runs giving maximum priority to real-time processes. For this reason, the server must be executed with administrator privileges. When a client process begins to be executed, the server transforms it into a real-time process, with the second highest priority in the system. Then, this client process is monitored and, if it exceeds the time slice determined for this period, it can either be halted or have its priority reduced to the minimum allowed for common process class until the following period begins. The decision to either halt the process or reduce its priority is based on the work-conserving parameter informed by the reservation or adaptation request.

Process monitoring is all done through real-time alarms that invoke a decision thread upon expiring. This thread has an implementation guided by the decisions illustrated in figure 2.

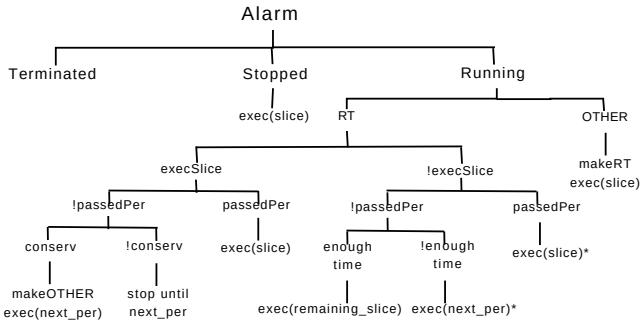

Figure 2: Decision steps of the alarm-treating thread.

When an alarm expires, the thread verifies the status of the process that requested the alarm. If its status is terminated, the process is removed from the process line in the server. If the status is stopped, this means that in the previous period the process has executed its processing time slice, was stopped until the end of the period because it was not work-conserving, and must execute a new time slice in the new period. If the process status is running, one must verify whether it is being executed as a real-time (RT) process or as a common one (OTHER). If the process is of type OTHER, this means the alarm expired because the process has ended its period. Therefore, the process must be turned again into a real-time one and be given a new time slice for execution.

If the process was running in real time, one must check whether its time slice has already been executed. If it has, one must verify if its period has expired. If it has, the process can run again during another time slice. If the period has not been exceeded, one must check whether the process is a work-conserving one. If it is work-conserving, it is transformed into a common, low-priority process and allowed to run with this status. Finally, if the process is not work-conserving, it is halted until the next period.

If the process has been running in real time but still did not finish its time slice, one must check whether its period has been exceeded. If it has, this means an error has oc-

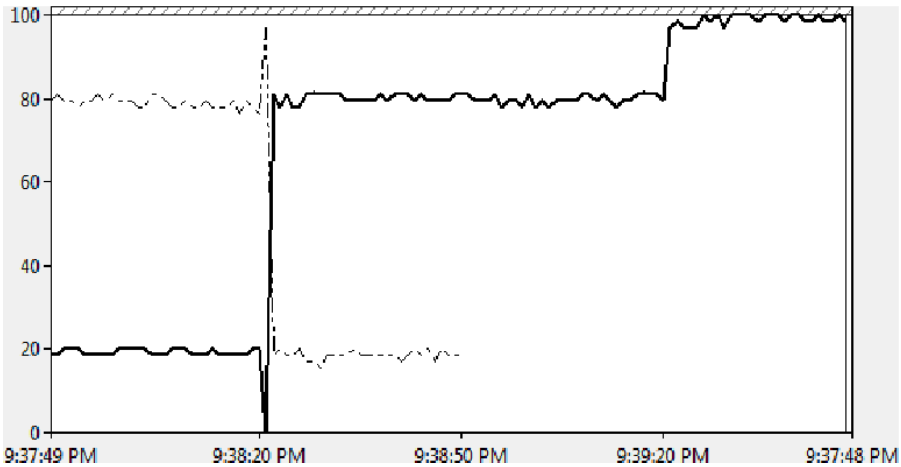

Figure 3: Performance of applications with and without reservation.

curred (marked with a star in figure 2). This error indicates that there are more reservations than the available processors can handle. In this case, the value of variable RESERVATION_LIMIT must be reduced. To avoid interrupting the server's execution, the process is allowed to continue running despite the error, restarting a time slice. If the period for the process has not expired, the process is allowed to run for the remaining time slice – if it is still possible to run what is left of the slice before the period expires – or for the remaining period – if an error similar to the one described previously has occurred.

4. EXPERIMENTAL EVALUATIONS

In order to evaluate CPUReserve's effectiveness, we executed two CPU-intensive applications in a same machine – one application without reservation, that is, being serviced in a best-effort way, and one application being executed with reservation guaranteed by CPUReserve. Initially, as one can see in figure 3, the latter application (solid line) has 20% of the CPU assigned to it. Consequently, the former application (dashed line) consumes the remaining machine's processing capacity, that is, 80% of the CPU. At 9:38:20, the second application suffers an adaption which increases its reservation to 80% of the CPU. As a result, the first application has its CPU consumption decreased to 20%. After 30 seconds, the former application is killed, releasing part of the machine's processing capacity. Nevertheless, the second application does not use the idle resource since it is configured to be non-work-conserving. Finally, after another 30 seconds, we changed this feature and the application started to consume all the available machine processing.

Tests were also carried out in order to evaluate CPUReserve's scalability and its use in a real distributed shared computing scenario. The tests have shown that the server provides significant scalability, but they have also revealed some limitations of the proposed system, as will be described in the following sections.

4.1 Scalability Tests

The scalability tests were performed in an Intel Centrino computer with 1.66 GHZ, 1G memory, running the Linux Operating System with kernel 2.6.24[4]. The goal was to estimate the amount of resources the server consumes to support an increasing number of processes. To achieve this, the CPUReserve server was initialized to manage both processors in the machine. The processes consisted in busy-waiting applications. The clients specified that the server should reserve them $10ms$ out of every $1000ms$, i.e. 1% of the CPU for each client process. We opted for a small percentage of the processing in order to set off many alarms in the smallest possible time[5]. This way, the server would work closer to its maximum capacity, as it would have to generate many threads for reservation treatment. The experiments have shown that, on its own – without responding to any client request –, the server consumes virtually 0% of CPU, 18 MB of virtual memory, and 732 KB of resident memory. CPU consumption increase linearly as the number of clients increases. The amount of CPU per process in the tests has been around just 0.2%.

Although tests with the server have shown that CPUReserve consumes few processing capacity, it was not possible to make experiments with over 50 clients. In such cases, the Operating System's response time suffered delays, leading to errors in the applications that were running on it, including some client processes which, instead of being executed with only 1% of CPU, were being executed with 3% or 4%. This reservation malfunction has two reasons: the large amount of time spent creating new alarm-treating threads, and the excessive amount of work the server must carry out at each $10ms$ interval – when the number of processes reaches 55, the server is no longer able to manage the priorities and temporizers of all processes whose period or time slice expire within a $10ms$ interval, causing some processes to continue running for longer than they should.

In order to evaluate the impact of reservation's requirements on the server, new tests were carried with different periods and slices: $20ms$ out of every $1000ms$, $40ms$ out of every $1000ms$, and $20ms$ out of every $2000ms$. The results are depicted in figure 4. Note that CPU consumption decreases

[4]A Windows version is also available but not evaluated here.
[5]The execution time of a process in CPUReserve is obtained by its file /proc/pid/stat. Time is given in jiffies, which, in the environment used for the tests, corresponds to $10ms$.

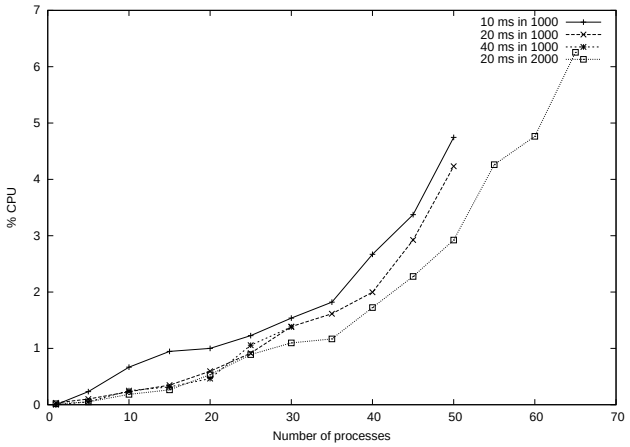

Figure 4: CPU usage with a varying number of processes and reservation specifications.

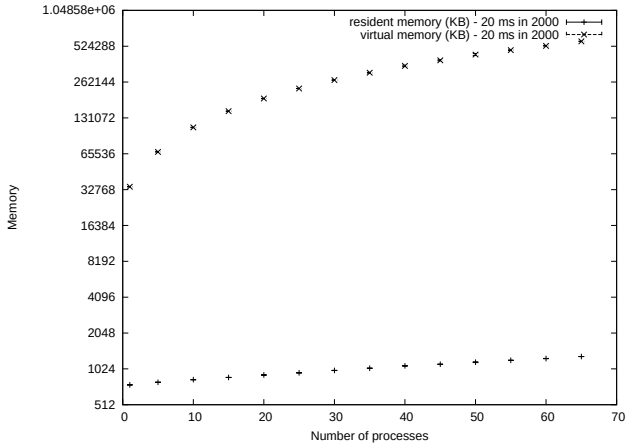

Figure 5: Virtual and resident memory usage for the server with a varying number of processes.

with 20-in-1000 time slices because, at this rate, the server treats each process in periods of around $20ms$. When the number of clients increases, this period gets smaller because there may be times when a process cannot execute $20ms$ within $20ms$, making necessary the setting of new timers and the creation of new threads to treat these timers. The high number of setting of new timers and of creation of new threads explain why CPU consumption does not decrease with 40-in-1000 time slices. With this latter rate reservation, the server is able to manage until 30 clients, because a number of clients bigger than 30 saturates the executing machine.

When the 1% reservation is obtained from executing $20ms$ out of every $2000ms$, one can notice that a bigger number of processes, actually 65, can be handled by the server. In this case the increased time slice allows the server to make a bigger number of system calls to manipulate processes' priority and scheduling policies.

Figure 5 depicts the virtual and resident memory usage for the server when it deals with an increasing number of processes. As a consequence of increasing memory sharing as the number of processes increases, one can notice a discrete reduction in the amount of virtual memory allotted for each process. With 5 processes, this amount was of around $13594KB$ per process; with 60 clients, this amount was reduced to around $8796KB$.

Note that the amount of resident memory allotted for the server is reduced as the number of processes increases. With 10 processes, this amount was of around $83KB$ per process; with 60 processes, this amount was reduced to around $21KB$. There are three reasons for this reduction: first because managing new processes through the server implies creating light processes such as threads, which consume few system resources; second because the amount of memory required to load the server has been divided into an increasing number of processes; and third because the short time interval between each client period caused some monitoring alarms to expire at the same time, being treated as a single thread.

Since the server has consumed a small amount of memory and less than 10% of CPU to manage a maximum of 65 clients, even considering the CPU consumption standard deviation, a maximum value of 0.9 can be assured for variable RESERVATION_LIMIT. Although, to take into account bigger deviations, this value is set to 0.8 in CPUReserve's code.

4.2 Limitations

Because CPUReserve was developed at user level, it is limited by the machine's Operating System. For instance, the Operating System has to provide the interfaces for system calls responsible for configuring processor binding and for changing process priorities and current scheduling policies. Also due to the fact that CPUReserve runs at user level, it can suffer interference from other priority-changing processes in the applications being managed (nice-type commands, for example).

Server execution can be unpredictable when a machine faces high workload. In these cases, an alarm might expire at a moment when its treating thread cannot be called. To prevent this, it is important to configure variable RESERVATION_LIMIT so that the Operating System does not suffer delays in its execution.

5. CPURESERVE AND VIRTUALIZATION

While a virtual machine is being created, it is possible to specify how much memory and disk the machine will be allotted. Sometimes, it is also possible to specify how much processing the machine can consume. It is the case of Xen monitor [4], for example. On the other hand, KVM (Kernel based Virtual Machine) does not provide this functionality [10]. In cases like KVM, one can ensure virtual-machines' performance isolation using CPUReserve to trigger a new VM. It is only necessary to have a CPUReserve server running on the host and to invoke a CPUReserve client request reservation with the command *kvm my_kvm_img <kvm_optional_args>*.

Although performance isolation can be ensured using virtual machines, the cost of managing them is high, and can deem the practice of isolating application executions by placing them in different virtual machines unfeasible [5]. Moreover, running more than one application per virtual machine no longer ensures performance isolation. An alternative is to envision hybrid isolating environments, where virtual ma-

chines coexist with user-level reservation managers.

Figure 6 presents a hybrid environment with a reservation hierarchy, where a physical machine is represented by two virtual machines. In the first one, an instance of CPUReserve is in charge of closely managing processing reservation among the applications of that virtual machine. Several applications may run at the same time in this machine without having their performance affected. For instance, the first machine can be set to be executed with 50% of a CPU and application1 with 30% of those 50%, i.e. with around 15% of a CPU's processing capacity.

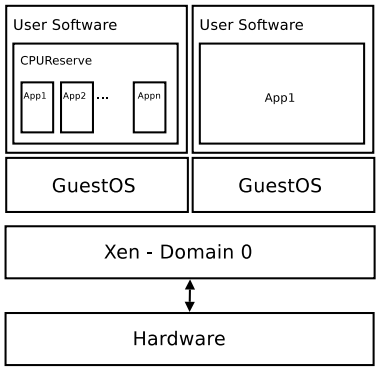

Figure 6: Processing reservation hierarchy in virtual machines.

To validate the proposal for hybrid environments, two prototypes were built with kernel 2.6.24. The first one was enabled with KVM module. In this scenario, CPUReserve were used to ensure each virtual machine performance isolation and to ensure applications' isolation inside each virtual machine.

The second prototype was built with Xen 3.2 enabled with SEDF scheduler. This prototype tried to reproduce the scenario illustrated in figure 6, where a virtual machine with processing capacity limited by the hypervisor isolates its applications performance using CPUReserve. The experiments with the prototypes have shown that CPUReserve is a light, easy-to-use manner to ensure performance isolation in virtualized environments. CPUReserve can be used with different virtualization techniques and for different levels of granularity since it can reserve processing for VMs and for applications inside VM's.

6. CONCLUSION

This paper has described the implementation of a user-level processing reservation system based on ideas proposed by DSRT [3]. Differently from traditional approaches found in the literature, the resulting system, called CPUReserve, does not require recompiling the Operating System's kernel and does not overload the server even when many clients are being monitored. The scalability presented in the tests, as well as its client-server architecture, are important features that make CPUReserve fit to be used in distributed shared computer environments. With this in mind, authors of CSBase, a framework for managing application execution in distributed environments [8], have been studying the use of CPUReserve to restrict the amount of CPU an application can use. Also in project CSBase, CPUReserve was used to measure some benchmarks accuracy.

Besides reserving time slices, CPUReserve also allows reserving CPUs in multi-processed machines. This feature was useful to test different reservation options in the system, as it was possible to saturate one of the machine's processors and see how the server behaved in a situation in which the Operating System could not function due to excessive workload. We expect this processor-managing feature to be useful in further scalability tests.

Finally, CPUReserve's implementation allows the separation between reservation policies and the reservation mechanisms, which facilitates the replacement of reservation policies. Developing new policies, using machine learning techniques to parametrize CPUReserve processes and inserting new kinds of reservations, like disk and memory, consist in further studies to be developed.

7. REFERENCES

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58, New Orleans, USA, 1999. USENIX Association.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP '03*, pages 164–177, New York, USA, 2003. ACM.

[3] H.-H. Chu and K. Nahrstedt. A soft real time scheduling server in unix operating system. In *Proceedings of IDMS '97*, pages 153–162, London, UK, 1997. Springer-Verlag.

[4] Citrix. Home of the xen hypervisor, Aug. 2008.

[5] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In M. van Steen and M. Henning, editors, *Proceedings of Middleware '06*, volume 4290 of *Lecture Notes in Computer Science*, pages 342–362. Springer Berlin / Heidelberg, 2006.

[6] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *Proceedings of GRID '04*, pages 34–42, Washington, USA, 2004. IEEE Computer Society.

[7] C. Lee, R. Rajkumar, and C. Mercer. Experience with processor reservation and dynamic QoS in real-time mach. In *Proceedings of Multimedia Japan 96*, Japan, Mar. 1996.

[8] M. J. Lima, C. Ururahy, A. L. de Moura, T. Melcop, C. Cassino, M. N. dos Santos, B. Silvestre, V. Reis, and R. Cerqueira. Csbase: A framework for building customized grid environments. In *Proceedings of WETICE '06*, pages 187–194, Washington, USA, 2006. IEEE Computer Society.

[9] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of RTAS '99*, page 111, Washington, USA, 1999. IEEE Computer Society.

[10] RamiTamir. Kernel based virtual machine, June 2008.

[11] S. Santhanam, P. Elango, A. Arpaci-Dusseau, and M. Livny. Deploying virtual machines as sandboxes for the grid. In *Proceedings of WORLDS '05*, San Francisco, USA, Dec. 2005.