Master in Computers Engineering Final Project
Year 2009-2010

Xen Scheduling with Load Balancing on Speed

Author:
Marco Antonio Gutierréz Giraldo

Professor in charge:
Manuel Prieto Matías

External direction contributor:
Juan Carlos Sáez Alcaide

Master in Computer Science Research
Faculty of Computer Science
Complutense University of Madrid

License

Xen is a Virtual Machine Monitor (VMM) originally developed by the Systems Research Group of the University of Cambridge Computer Laboratory, as part of the UK-EPSRC funded XenoServers project. Xen is freely-distributable Open Source software, released under the GNU GPL v2 [1] . Code generated and distributed from this project is released under the GNU GPL v3 license terms.

Licensing Exceptions (the relaxed BSD-style license) [2]

For the convenience of users and those who are porting OSes to run as Xen guests, certain files in the Xen repository are not subject to the GPL when distributed separately or included in software packages outside the repository. Instead Xen specifies a much more relaxed BSD-style license. Affected files include the Xen interface headers, MiniOS (extras/mini-os) and various drivers, support functions and header files within Xen-aware Linux source trees. In all such cases, license terms are stated at the top of the file or in a COPYING file in the same directory. Note that_any_ file that is modified and then distributed within a Linux kernel is still subject to the GNU GPL.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the page of the GNY project in the section entitled "GNU Free Documentation License"[3].

Authorization

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Xen Scheduling with Load Balancing on Speed”, realizado durante el curso académico 2009-2010 bajo la dirección de Manuel Prieto Matías en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Marco Antonio Gutiérrez Giraldo

"A las aladas almas de las rosas
del almendro de nata te requiero,
que tenemos que hablar de muchas cosas,
compañero del alma, compañero."

Miguel Hernández

Acknowledgements

I want to thank Professor Manuel Prieto Matías and Juan Carlos Sáez Alcaide for their help and collaboration in the development of this project. The ArTeCS's Systems Administrators for such a good work and help provided.

I also wan to thank my family for their considerable patience and support not only during this project but throughout all my life.

Thanks to all of those who helped me get where I am today and those who tolerate me during all this time.

Thanks to the Xen community for their fabulous open-source virtualization solution, and specially those from the Xen-devel mailing list for their patient and inconsiderately help to noobs. Thanks to the Debian project for their package solutions, their availability and great documentation. Also thanks to all the free software community, specially to Richard Stallman for such a good idea.

And finally also thanks to Dietrich Mateschitz for giving me wings and all the energy needed to complete this project.

Contents

1 Xen Hypervisor **7**
1.1 Introduction 7
1.2 History 7
1.3 Xen Virtualization Solutions 8
1.3.1 Paravirtualization 8
1.3.2 Hardware assisted virtualization 10
1.4 Architecture 11
1.4.1 Domain 0 12
1.4.2 Unprivileged Domains (domU) 13

2 Scheduling on Xen **15**
2.1 Scheduler development 16
2.2 Xen Schedulers 19
2.2.1 Roundrobin 19
2.2.2 Borrowed Virtual Time 19
2.2.3 Atropos 20
2.2.4 SEDF 21
2.2.5 Credit Scheduler 21
2.2.6 Credit Scheduler 2 24

3 Load Balancing on Speed **27**
3.1 Load Balancing 28
3.2 Load Balancing HPC applications 29
3.3 Motivation For Speed Balancing 30
3.4 Algorithm 33
3.5 Speed Balancing Implementation on Xen 34
3.5.1 Algorithm and Structures 35

4 Experiments and results **39**
4.1 Experimental Setup 39
4.1.1 Virtualization environment setup 40
4.2 Performance results 42
4.2.1 Baseline comparision 42

4.2.2 Performance impact due to sharing CPUs 43
4.2.3 Adjusting the spin threshold 44
4.2.4 SPEC jbb 2005 48
4.2.5 Increasing number of shared cores 50

5 Conclusions and Future Work **53**
5.1 Future Work 53

A The Load Balancing on Speed Pseudocode **55**

Bibliography **57**

Figures Outline **i**

Resumen

Hoy en día son muy frecuentes las máquinas con gran cantidad de cores. Es muy probable que el grado de paralelismo se vea incrementado en un futuro próximo. Hacer un uso completo de este hardware es una tarea difícil de conseguir. Los entornos de virtualización ofrecen hoy en día una buena aproximación para poder sacarle el máximo partido a este tipo de hardware. Una planificación adecuada, y en consecuencia un balanceo adecuado, de las máquinas virtuales de estos entornos es una tarea importante y difícil de lograr adecuadamente.

Aquí proponemos el uso del algoritmo de *Load Balancing on Speed* (Balanceo de Carga basado en velocidad)[4] con el fin de acelerar las máquinas virtuales que ejecutan aplicaciones paralelas. De esta manera el sistema obtendrá un mejor rendimiento global. La técnica de *Load Balancing on Speed* está diseñada específicamente para aplicaciones paralelas que corren en sistemas multicore. Los cores se clasifican en *rápidos* y *lentos* de acuerdo con los parámetros de las VCPUs (cpus virtuales). Nuestro algoritmo balancea el tiempo que una VCPU se ha ejecutado en cores *rápidos* y *lentos*.

Hemos implementado y probado nuestro algoritmo en el sistema Hipervisor Xen[5]. A continuación se presenta información básica sobre este Hipervisor y sus algoritmos de planificación, así como nuestra propuesta y correspondientes resultados. Estos resultados se han obtenido utilizando un cierto escenario de virtualización y discutiendo el comportamiento de una gran variedad de cargas de trabajo corriendo en este entorno.

Los resultados han demostrado que nuestro algoritmo es rentable. *Load Balancing on Speed* parece mejorar el rendimiento con respecto al planificador por defecto de Xen (el planificador de créditos, llamado Credit Scheduler). Varios valores de spin se discuten junto con su rendimiento en nuestro entorno de virtualización. También se comentan algunos pequeños cambios que podrían ser implementados con el fin de obtener los mejores beneficios de planificación de este algoritmo de balanceo de carga.

Palabras clave: Xen, Hipervisor, Virtualization, Programación Paralela, Balanceo de Carga, Speed Balancing, Multicore

Abstract

Heavily multicored machines are prevalent nowadays. The degree of parallelism is likely to be highly increased in the near future. Making full use of this hardware is a hard task to achieve. Virtualization environments offer nowadays a good approach in order to make the most of this hardware. Properly scheduling, and therefore properly balancing, the virtual machines within this environment is an important and hard task to properly achieve.

Here we propose the use of Load Balancing on Speed algorithm [4] in order to speed up the Virtual Machines executing parallel applications. This way the system will achieve a good overall performance. The load balancing technique is designed specifically for parallel applications running on multicore systems. Cores are classified as *fast* and *slow* according to parameters from the running VCPUs Our algorithm balances the time a VCPU has executed on *faster* and *slower* cores.

We have implemented and tested our algorithm in the Xen Hypervisor system [5]. Some background information about this Hypervisor and its baseline scheduling algorithm is presented along with our proposal and performance results. These results have been obtained using a certain virtualization scenario and disscussing behavior across a variety of workloads running on that environment.

Results have shown our algorithm to be profitable. Load Balancing on Speed seems to improve performance over the Xen default's scheduler (Credit Scheduler). Several spin values are discused along with their performance in our virtualization environment. We also discuss are some tunings that could be performed in order to obtain the best scheduling benefits from this load balancing algorithm.

Key words: Xen, Hypervisor, Virtualization, Parallel Programming, Load Balancing, Speed Balancing, Multicore

Chapter 1

Xen Hypervisor

1.1 Introduction

Xen is an open-source para-virtualizing virtual machine monitor, also known as hypervisor, for the x86 processor architecture. Xen isolates virtual machines in domains executing several in a single physical system with close-to-native performance, thanks to the para-virtualization technology. It is able to run any of the x86/32, x86/32 with PAE and x86/64 platforms. Xen also supports Intel Virtualization Technology (VT-x) for unmodified guest operating systems, like Microsoft Windows. It supports almost all Linux device drivers, live migration of running virtual machines between physical hosts and up to 32 virtual CPUs per guest virtual machine [6].

1.2 History

Xen was originally developed by the Systems Research Group at the University of Cambridge Computer Laboratory as part of the XenoServers project, funded by the UK-ESRC.

Xenoservers aim to provide a "public infrastructure for global distributed computing". That's the reason for Xen to play a key part in allowing efficient partitioning of a single machine to enable multiple independent clients running operating systems and applications in an isolated environment. The project web page contains further information along with other technical reports.

Xen was first described in a paper presented at the ACM Symposium on Operating Systems Principles (SOSP) in 2003 [5], and the first public release, the 1.0 version, was made October of that same year. Nowadays Xen has grown into a fully-fledged project in its own right, enabling us to investigate interesting research issues regarding the best techniques for virtualizing resources such as CPU, memory, disk and network. Some of the main project contributors are XenSource, IBM,

Intel, AMD, HP, RedHat, Novel... many improvements have taken place, such as scalability, support and performance to come to the last release, Xen 4.0.1, the one used in this project.

1.3 Xen Virtualization Solutions

Although x86 is difficult to virtualize, it is so widespread and popular for business use that much effort has been put into getting around those limitations intrinsic to the platform and several solutions have been proposed. Some approaches like VMWare's old binary rewriting have a nice benefit allowing the virtual environment to run in userspace, but imposes a performance penalty. Xen offers two solutions to achieve a good virtualization performance, the paravirtualization the Hardware-Assisted Virtualziation.

1.3.1 Paravirtualization

Paravirtualization is an approach that tries to obtain the closest system to x86 that we can virtualize. The environment that is presented to a Xen guest has its differences with that of a real x86 system. However, it is similar enough in that it is usually a simple task to port an operating system to the Xen platform.

The buggiest difference, from the operating system point of view is that, in the Xen system, it runs in ring 1, instead of running in ring 0 (which is the case of a non-virtualized OS), as shown in Figure 1.1. That means that it is not allowed to perform any privileged instructions. Instead the Xen hypervisor provides the virtual machine with a set of hypercalls that corresponds to the privileged instructions.

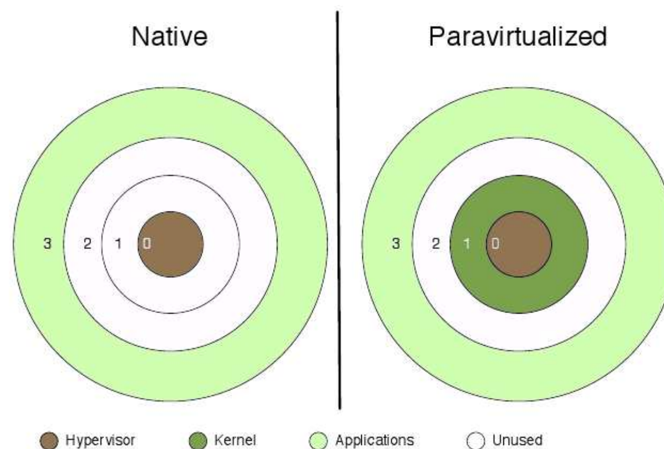

Figure 1.1: Ring Usage in native and paravirtualized systems

The hypercalls works in a very similar manner to system calls, but they are managed

by the hypervisor instead of the OS. The main difference is that they use a different interrupt number.

In Figure 1.2 you can see the main differences are at the ring transitions when a system call is issued from an application running in a virtualized OS. Here, the hypervisor, not the kernel, has interrupt handlers installed. Thus, when interrupt is raised, execution jumps to the hypervisor, which then passes control back to the guest OS. This extra layer of indirection imposes a small speed penalty, but it does allow unmodified applications to be run. Xen also provides a mechanism for direct system calls, although these require a modified libc.

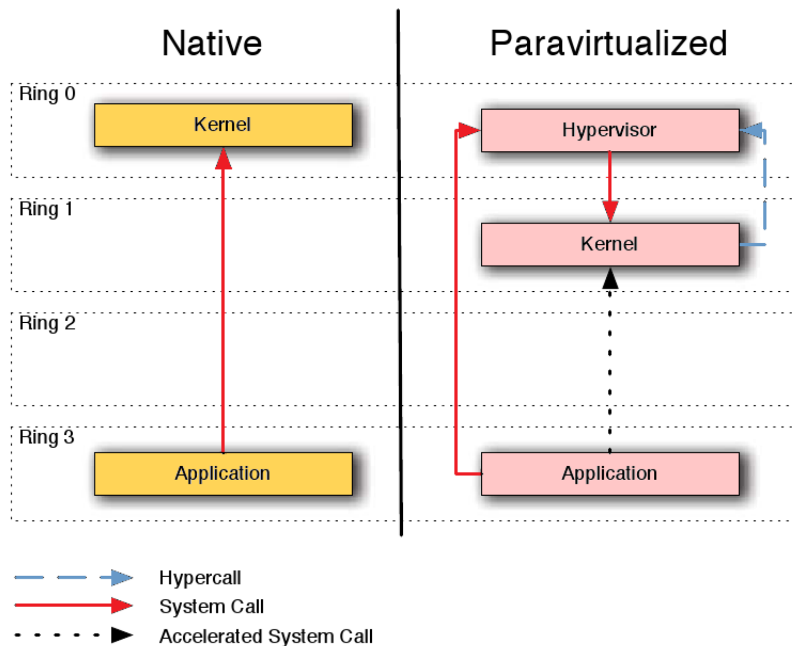

Figure 1.2: System calls in native and paravirtualized systems

As GNU/Linux, Xen uses the MS-DOS calling convention, rather than the UNIX convention used by BSD systems. Note that Xen, like Linux, uses the MS-DOS calling convention, rather than the UNIX convention used by Free BSD. This means that parameters for hypercalls are stored in registers, starting at EBX, rather than being passed on the stack.

In the new versions of Xen, as the one we used, the hypercalls are issued by an extra layer of indirection. The guest kernel calls a function in a shared memory page (mapped by the hypervisor) with the arguments passed in registers. This allows more efficient mechanisms to be used for hypercalls on systems that support them, without requiring the guest kernel to be recompiled for every minor variation

in architecture. Newer chips from AMD and Intel provide mechanisms for fast transitions to and from ring 0. This layer of indirection allows these to be used when available.

1.3.2 Hardware assisted virtualization

Different processor manufacturers have extended the architecture in order to make virtualization considerably easier for x86. AMD and Intel have introduced in the chip several extensions that are now supported by the Xen virtualization environment.

AMD-V, also known as Pacifica [7] and Intel's extensions, known as Intel Virtualization Technology [8] are supported in the latest Xen Hypervisor release. The basic idea behind these is to extend the x86 ISA to make up for the shortcomings in the existing instruction set. Conceptually, they can be thought of as adding a "ring -1" above ring 0, allowing the OS to stay where it expects to be and catching attempts to access the hardware directly. In implementation, more than one ring is added, but the important thing is that there is an extra privilege mode where a hypervisor can trap and emulate operations that would previously have silently failed.

When compared to paravirtualization, hardware assisted virtualization, often referred to as HVM (Hardware Virtual Machine) [9], others some trade-os. It allows the running of unmodified operating systems. This can be particularly useful, because one use for virtualization is running legacy systems for which the source code may not be available. The cost of this is speed and flexibility. An unmodified guest does not know that it is running in a virtual environment, and so can't take advantage of any of the features of virtualization easily. In addition, it is likely to be slower for the same reason.

Nevertheless, it is possible for a paravirtualization system to make some use of HVM features to speed up certain operations. This hybrid virtualization approach offers the best of both worlds. Some things are faster for HVM-assisted guests, such as system calls. A guest in an HVM environment can use the accelerated transitions to ring 0 for system calls, because it has not been moved from ring 0 to ring 1. It can also take advantage of hardware support for nested page tables, reducing the number of hypercalls required for virtual memory operations. A paravirtualized guest can often perform I/O more efficiently, because it can use lightweight interfaces to devices, rather than relying on emulated hardware. A hybrid guest combines these advantages.

Xen-HVM has device emulation based on the QEMU project to provide I/O virtualization to the virtual machines. The system emulates hardware via a patched

QEMU "device manager" (qemu-dm) daemon running as a backend in dom0. This means that the virtualized machines see as hardware: a PIIX3 IDE (with some rudimentary PIIX4 capabilities), Cirrus Logic or vanilla VGA emulated video, RTL8139 or NE2000 network emulation, PAE, and somewhat limited ACPI and APIC support and no SCSI emulation.

1.4 Architecture

Xen works between the OS and the hadware it provides a virtual environment in which a kernel can run. There are three main components in a Xen system: the hypervisor, kernel and userspace application.

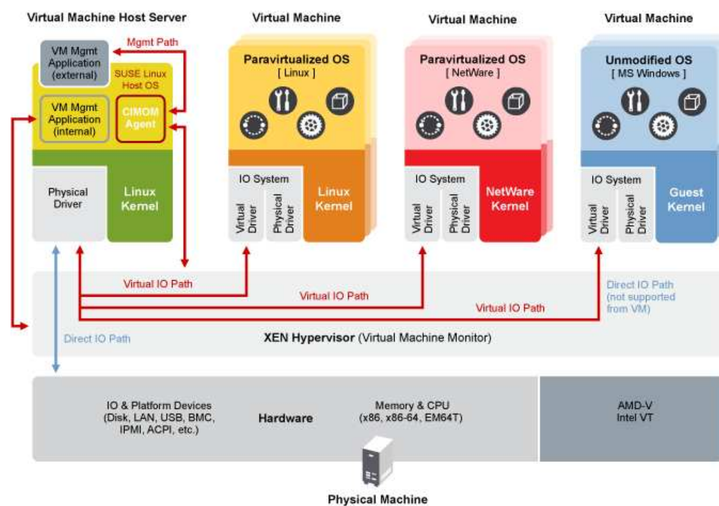

Figure 1.3: Xen Architecture Overview. Illustration from Novell.com

The Xen system includes several layers. The Xen Hypervisor runs as the lowest and most privileged one Xen may host multiple guest operating systems, each of which is executed within a secure domain, which encapsulates a complete running virtual environment. Domains are scheduled by Xen to make effective use of the available physical CPUs. Applications are managed by each OS, scheduling each application within the time allotted to the domain by Xen.

Figure 1.3 shows an overview of the Xen system and how layers are placed. The layering is not absolute, as you can see not all guests are created equal, depending in the virtualization solution and the OS. Xen differentiates two main groups of domains, privileged and unprivileged. One special domain guest, domain 0 has several privileges over the others in order to be able to perform management tasks.

1.4.1 Domain 0

At the Xen boot, on of the first things to happen is that the Domain 0's guest kernel gets load. This is typically specified in the boot loader as a module, and so can be loaded without any filesystem drivers being available. Domain 0 (dom0) is the first guest to run and it also has some privileges. Domain 0 has no device drivers and no user interface on its own. All of it is provided by the operating system and userspace tools running in the dom0 guest. Usually this domain is a GNU/Linux system, although it could also be a BSD or Solaris system.

The rest of the domains are referenced to as domain U or domU (which stands for domain-Unprivileged). However it is also possible in the latest releases of Xen to let some privileges to other domains different from dom0.

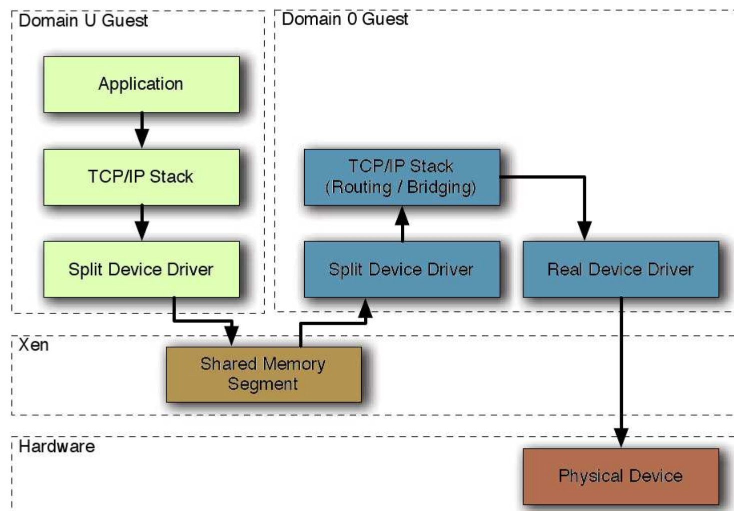

Figure 1.4: A network package sent from an unprivileged domain

The main task performed by the dom0 guest is to handle the different devices. As this guest runs at a higher level of privilege than the rest it is capable to get direct access to hardware. Part of the responsability of handling devices is multiplexing them for the virtual machines, mainly because up to know most of the hardware does not natively support being accessed by multiple operating systems.

An example of dom0 management behavior is shown in Figure 1.4. It shows the path of a packet when it is sent by an application in domU guest. The bottom of the TCP/IP stack is not a regular network interface driver, instead it is a piece of code that leaves the packet in some shared memory [10]. The rest of the driver, running on the dom0 guest, will read the packet and insert it to the firewalling rules of the operating system where it will make it down to the physical network device.

1.4.2 Unprivileged Domains (domU)

Domain Us are the unprivileged and therefore restricted domains of Xen. Usually domUs are not allowed to perform any hypercalls directly accessing hardware, although in some special cases access to one or more devices is granted.

DomU implements the front end of split device drivers. At a minimum, it is likely to need the XenStore and console device drivers. Most also implement the block and network interfaces. Because these are generic, abstract, devices, a domain U guest only needs to implement one driver for each device category. For this reason, there have been a number of operating systems ported to run as domain U Xen guests, which have relatively poor hardware support when running directly on real hardware. Xen allows these guests to take advantage of the Domain 0 guest's hardware support.

You are only allowed to have one dom0 guest but you can have an arbitrary number of domU guests on a single machine, and they may be able to be migrated (largely depending on the configuration).

The differences betwen dom0 and domU are sometimes not so much. It is possible to let a domU guest access directly the hardware, and even to host split device drivers.

Chapter 2

Scheduling on Xen

Xen System is fully multitasking. The hypervisor is the one in charche of ensuring that every guest running gets its shared of CPU time. In the same way that a multitasking operating system works, scheduling in Xen is a tradeoff betwen achieving fairness for running domains and achieving good overall throughput.

Due to the nature of some of the uses of the Xen hypervisor some other constrains are applied. One common use of an hypervisor environment is to provide virtual dedicated server for a wide variety of customers. They might have some form of service level agreement associated with them, and therefore it might be required to assure that no customer receives less than his previously allocated amount of CPU, and in the the case that he receives more, keep track of it [11].

There are many concepts and ideas that scheduling for an operating systems that provides an N:M threading library shares with scheduling for systems like Xen. In an operating systems the kernel schedules N threads, usually one per physical context), which a userspace library multiplexes into M userspace threads. Virtual CPUS (VCPUS) from Xen could be analogous to kernel threads and the userpace threads reperesent the processes within the domain.

There might be another layer within a Xen system, because the guest domains might have an userpace itself and the corresponding threads running on top of it. This comes down to the fact that there might be up to three different schedulers between an userspace thread in the virtual machine and the physical CPU:

1. The userspace threading library mapping userspace threads to kernel threads

2. The guest kernel mapping threads to VCPUs

3. And the hypervisor mapping VCPUs to physical CPUs (the one we have developed in this project)

The bottom of the stack, the Xen hypervisor scheduler, needs a predictable behavior. The schedulers above it will make certain assumptions on the behavior of the underlying scheduling, and if these assumptions are not properly predicted they will come up with highly suboptimal decisions. This will lead to bad, or unpredictable behavior for processes in the running domains. The design and tuning of a good scheduler is one of the most important factors in keeping a Xen system running well.

There can only be a scheduler running for the Xen hypervisor at a time, and the desired one must be selected at boot time by specifying an argument to the hypervisor. The hypervisor reads the parameter "sched" from the boot parameters and looks for its match in the opt name field of the interface (see Figure 2.1).

Through our research we have been trying to improve the current Xen scheduler in order to make it more workload aware, and able to adapt to the future multicore environments. We have developed a better way for balancing the load in the credit scheduler

2.1 Scheduler development

Hypervisor's schedulers in Xen are developed following a strict interface, where you can map your scheduling functions. You can find it in the scheduler.c file within the Xen's source code. Listing 2.1 shows this interface. It is defined by a structure containing pointers to functions that are used to implement the functionality of the scheduler. This interface provides the developer with an abstraction layer to implement different scheduling policies.

```
1  struct scheduler {
2      char *name;              /* full name for this scheduler      */
3      char *opt_name;          /* option name for this scheduler    */
4      unsigned int sched_id;   /* ID for this scheduler             */
5
6      void          (*init)           (void);
7
8      int           (*init_domain)    (struct domain *);
9      void          (*destroy_domain) (struct domain *);
10
11     int           (*init_vcpu)      (struct vcpu *);
12     void          (*destroy_vcpu)   (struct vcpu *);
13
14     void          (*sleep)          (struct vcpu *);
15     void          (*wake)           (struct vcpu *);
16     void          (*context_saved)  (struct vcpu *);
17
18     struct task_slice (*do_schedule) (s_time_t);
19
20     int           (*pick_cpu)       (struct vcpu *);
21     int           (*adjust)         (struct domain *,
```

```
22                                        struct xen_domctl_scheduler_op *);
23     void          (*dump_settings)   (void);
24     void          (*dump_cpu_state)  (int);
25
26     void          (*tick_suspend)     (void);
27     void          (*tick_resume)      (void);
28 };
```

Listing 2.1: Xen Hypervisor interface for schedulers

When coming up with a new scheduler, with its own policy, it is necessary to implement one of these structures pointing to the new implemented scheduling functions, and to add it to a static array of available schedulers so Xen itself can get aware of it.

It is not necessary that all the functions defined in the structure get defined for any give scheduler. If the function it's not implemented, it should be initialized with a NULL pointer and it will be simply ignored. An implementation of a simple scheduler could set almost all the functions to NULL and the system will still work, not with a good performance so it might not be very useful. Functions are called via a macrom. This macrom tests for a non-NULL value and it will return 0 if one is found. However, not all the functions can be set to null, for instance do_schedule() must be a valid function that picks a next VCPU to run. So basically a scheduler that does not implement this function will crash the hypervisor.

The do_schedule pointer will need to point to a real function. This function will receive the time and return a struct containing the next VCPU to run and the amount of time for which it should run before being preempted in a task slice structure.

In order to be able to return the VCPU in this function, the scheduler should keep a record of which VCPUs are available at that time. When a VCPU is started, first goes through the VCPU initialization function. The scheduler should use this call in order to keep track of the created VCPUs in the system and their availability at a given time.

Along with the list of available schedulers, scheduler.c contains all of the scheduling-independent code. This file contains analog functions to all of those defined within the scheduler interface. This allows the system to perform several general operations before calling the functions implemented in the scheduler, if they exist. If we look at the function schedule(), is the fined in schedule.c where it first deschedules the running domain and then calls the linked schedule function from the scheduler. The function from the scheduler will return a new task to schedule and the time which it should run. Then the code from schedule.c sets up a timer to trigger at the end of the quantum and begins running a new task.

The scheduler API, showed in Listing 2.1, contains four non optional fields. The

name field should contain a human-readable name for the new scheduler proper identification, however the opt_name will hold an abridged version in order to select the scheduler when specifying so to the hypervisor by the "sched" option at boot time.

The definition of the VCPU structure contains a sched_priv pointer. This pointer should be used to store the private information from the scheduler regarding that VCPU. At the init_vcpu the required space memory for this purpose should be allocated and initialized, so after that it can be used through this pointer It is also a task of the scheduler to destroy it when is not needed anymore, typically when the destroy_vcpu function is called.

There are also a physical CPU and a domain structure. Some schedulers may need to differentiate the VCPUs by their owner domains so they can be treated differently. The domain structure contains a VCPU field with an array of the virtual CPUs owned by that domain. The scheduler can use this to ensure fairness between domains, instead of just between VCPUs. It is also important to know that not all VCPUs for a certain domain might need to be run at the same speed or even at the same moment. The administrator might want to delegate all scheduling to Xen, so he will create one VCPU per process in a guest domain. The guest's scheduler is then only responsible for assigning tasks to VCPUs.

The dump functions are only there for debugging purposes. Administrators can request the current status of the hypervisor and this two functions will be called in order to dump the state of the scheduler.

The latest version of Xen only comes with two schedulers, the EDF old and simple version, called SEDF, and the newer Credit Scheduler which achieves better performance. That and because of the limitations of the SEDF scheduler new versions of Xen use the Credit Scheduler as the default option.

Modification to the hypervisor sources and recompilation is needed in order to add a new Scheduler. Generally, each scheduler can be separated out into its own source file, and the only modification required to the rest of the Xen sources is to add it to the list of available schedulers at the top of scheduler.c. This makes it relatively easy to maintain a scheduler outside the main Xen tree.

Regular users can tune the provied scheduler's parameters to achieve a better system performance [12]. Credit Scheduler, for instance is highly configurable. This should be enough for any regular user of Xen. Figure 2.1 shows two virtual machines with same CPU allocation and web servers running on them. As you can see the throughput achieved is different depending on the scheduler selected and the weight values singed to the virtual machines.
However anyone, in a situation like us, could require scheduling beyond that provided by existing schedulers. That's why we have implemented and chosen the load

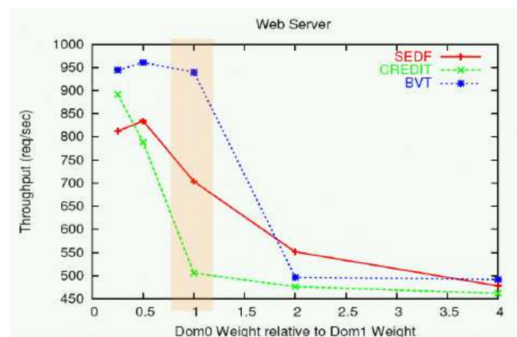

Figure 2.1: Schedulers behavior according to their weight

balancing on speed algorithm, which we think is a good choice and matches our necessities in order to obtain the results we want to achieve.

2.2 Xen Schedulers

Realtime scheduling is a difficult problem for Xen, due to the nature of virtualization. It is somewhat difficult to design a realtime scheduler on a single machine without compromising throughput too much. The two (or sometimes three) tier nature of Xen makes it even harder. A solution would likely require some close cooperation between the hypervisor and kernel schedulers, with the kernel scheduler registering wake-up deadlines with the hypervisor. Even this, however, would only allow best-effort realtime scheduling. Soft realtime scheduling is important for a lot of tasks, particularly those on the desktop that involve media recording or playback.

Along the development of Xen several Schedulers have been includes [12], version 4.0.1, the last release and the one used here only includes SEDF and Credit Scheduler, being the last one the one used by default.

2.2.1 Roundrobin

The round robin scheduler was included in releases of Xen as a simple demonstration of Xen's internal scheduler API. It is not intended for production use. It had a fixed-length quantum that, in the absence of a guest voluntarily yielding its CPU time, each domain would be run for a fixed quantum in order.

2.2.2 Borrowed Virtual Time

The Borrowed Virtual Time (BVT) [13] scheduler was included in Xen 2. This scheduler uses the concept of virtual time. It only elapses while the domain is

scheduled. Wighting of VCPUs is done by increasing the virtual time at a configurable rate per-domain basis. domains with high weight would have bigger virtual time than one with a small weight after both spending the same amount of wall time running. The do_schedule function of this scheduler will pick the runnable VCPU with the earliest effective virtual time. This alone gives a fairly close approximation of a round robin scheduler. The feature that accounts for the word “borrowed” in the name is the capability of domains to “warp.” The administrator will have the responsability of setting a range of time within each domain will be able to set its virtual time to some point in the past.

There are two limits placed on warping: the maximum amount of time a VCPU can run warped for, and the maximum amount of time it can warp. The effective virtual time is calculated by subtracting the warp time from the actual time. This effective time and the virtual one are associated to a VCPU. A domain can enter the warping state, in case it needs a low processing latency, so the BVT scheduler will more likely select it to be scheduled, but this will only occur until its warp time has been elapsed. After that it will have to wait one of the configurable intervals to be allowed to warp once again.

2.2.3 Atropos

This scheduler comes also from Xen 2. It provides scheduling with soft realtime. Atropos guaranteed that every domain would run for n milliseconds every m milliseconds of wall clock time. This was a good approach for virtual machines with a high sensibility for latency, but it was not ideal for CPU throughput. A fixed portion of CPU time was guaranteed for each CPU, and the reminder was shared out evenly.

Workloads where a domain spends come of its time using almost no CPU and then some sing as much as it has available did not work well with the Atropos scheduler. Mainly because domain could not obtain CPU "bursts". This is due to the fact that each CPU was guaranteed a fixed allotment of CPU time, and the reminder was shared out evenly. Also, this scheduler was not able to allow overcommitting of CPU resources.

Atropos maintained a record of the end of each VCPU’s deadline and the amount of time it would be allowed to run before this, basically the amount of time it had to run every interval, minus the amount of time it had run this interval. A queue will maintain runnable domains ordered by deadline. Whenever the scheduler was invoked, it would go through all this steps:

1. Get the domain’s remaining runtime and substract the amount of time of which it has just run. If the resultant runtime is zero, the domain will be moved from the run queue to the waiting queue.

2. Domains in the waiting queue that are due to run again are moved back to

the run queue.

2.2.4 SEDF

The Simple Earliest Deadline First (EDF) [14] scheduler is one of the two schedulers present in the latest release of Xen. It is the oldest of the two, and probably the main candidate to be phased out some time in the future.

The EDF scheduler sets each domain to run for an n milliseconds slice every m milliseconds. The values of n and m are configurable by the administrator on a per-domain basis. When the do_schedule() function is called it will choose to run the VCPU which has the closest deadline.

Let's consider these there domains running over a Xen Hypervisor with a SEDF scheduler:

1. 90ms slice every 500ms
2. 20ms slice every 50ms
3. 30ms slice every 50ms

SEDF scheduler works on VCPU's but for making it easier we will consider that each domain has only one VCPU. Initially, domains 2 and 3 have the earliest deadlines for starting their quanta, because they both need to be scheduled within 50 milliseconds. Domain 3 has the earliest deadline for starting its quantum, because it must be run in the next 20 milliseconds, whereas domain 2 can wait for 30 milliseconds.

After domain 3 has run it moves into the future its next deadline, beyond domain 2. These two scheduled periodically for around 410 milliseconds, until domain 1 has to be run. It will then take control of the CPU for 90 ms. The other two domains will not be able to run until the first one has ended, and taking into account that this is longer than the period of the other two domains to be runned, it is undeniable that they will miss their allotments.

In this case, the scheduler will detect it and treat it specially. Allowing domain 1 to run for the whole slice would lead to the other two VCPUs missing theirs, so the SEDF scheduler will reduce the allocation so that it terminates in time for the next deadline.

2.2.5 Credit Scheduler

On the latest, and our version of Xen, the Credit scheduler is default scheduler. If using the "sched" variable in the code is not matching any scheduler in the list of available schedulers in the file scheduler.c then it will use the Credit Scheduler. It is widely configurable for the administrator and it seems that will be used for

some time since the Xen developers are right now working on a new version of it, improving some aspects of its behavior.

In this scheduler each domain has two main properties associated with it, a weight and a cap. The weight determines the share of physical CPU time that the domain will get. The cap is mainly the maximum of CPU time the domain can get. As you can see in Figure 2.1 weights are relative to each other. It would have the same performance impact giving two domains weights of 2 and 3 respectively than giving them 200 and 300 because the relation between them is the same. However the cap is an absolute value which represent the portion of total CPU that can be used by a domain.

By default, Credit Scheduler is work-conserving, but the cap parameter is provided so the administrator can set it in order to force non-work-conserving mode. For instance, two virtual machines with priorities of 128 and 64, the second one will get half as much CPU time as the first if both of them are busy. In the case that the first goes is idle the second one will get the whole CPU. In the case that all domains in a system have a cap and the value of the sum of all together is below the total CPU capacity, the scheduler will not be running any domains for some of the time.

The Credit Scheduler uses a fixed-size 30ms quantum. At the end of each quantum, it selects a new VCPU to run from a list of those that have not already exceeded their fair allotment. If a physical CPU has no underscheduled VCPUs, it tries to pull some from other physical CPUs.

Whether a CPU is over or underscheduled depends on how it has spent its credits. Credits are awarded periodically, based on the priority. Consider the following example domains:

	Priority	cap
Domain 1	64	No
Domain 2	64	35%
Domain 3	128	No

Table 2.1: Example of System Configuration for Credit Scheduler

Again as in the previous scheduler's example each domain will have one VCPU for making it simpler. At the beginning of the scheduling interval the first two domains will have 64 credits and the last one will have 128. All the VCPUs have work to do, however they will be scheduled in a round robin manner. At some point the first two domains will run out of credits. At this point the VCPU from the last domain will get all of the CPU for itself until the scheduling interval is done.

Because first and second domains have the same number of credits if the last domain is idle, both of them will get the same amount of CPU. Eventually the second one will reach its cap of 35%. Then the first VCPU will continue running. This way, it will ends its allowance of credits, and in the next accounting process it will be moved to the overscheduled queue. At the same accounting process the other two VCPUs will be marked as underscheduled and therefore continue to gain credits.

In the time that domain 2 is capped, all new credit allocation that take place will take this into account, dividing the credits that where supposed to be for domain 2 between the rest of the domains in the system (domain 1 and domain 3). So administrators should be careful not to set the priority of a domain larger than the percentage of the CPU allocated to its cap, otherwise bad-performance results will take place.

A timer will tick every 10 ms a function in order to burn the corresponding credits of the running VCPUs, it will also cap the minimum number or credits as the number that would be achieved by a process running for one complete time slice having started with no credits.

This minimum value has a small effect on the scheduling algorithm. If a VCPU is getting enough runtime to be exceeding the minimum threshold, the rest of the VCPUs must be either capped or idle. This way the running VCPU will get all the credits when determining allocation of the credits, way more than it otherwise could, balancing out the drop. This is because the other VCPUs will be ignored in the credit allocation due to their capped or idle state. When the other VCPUs get back to work they will take part then in the credit allocation and the currently running VCPU will be throttled back to its fair share.

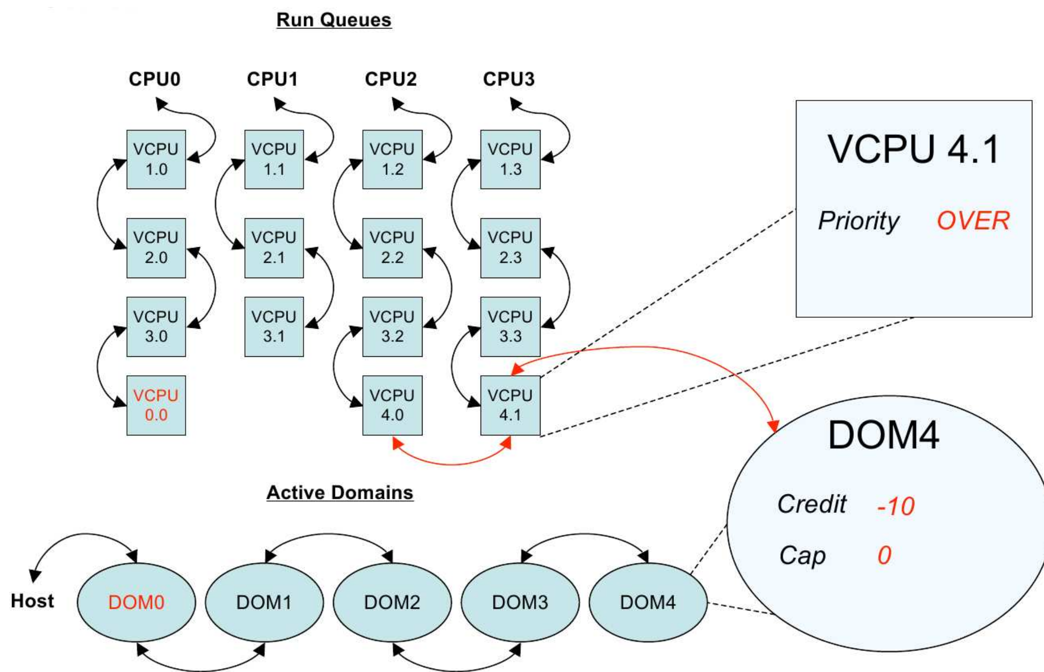

Figure 2.2: Example of a system running with a Credit Scheduler

In Figure 2.2 you can see the state of a system running with the credit scheduler. there are four physical CPUS, with their associate runqs. This runqs contain the VCPUs with their associated credit value. So this VCPUs will get scheduled following this information and the credits policy of the Credit Scheduler. There are, also, five active domains that own the VCPUs, with their credit and cap scheduler information associated.

2.2.6 Credit Scheduler 2

A new credit scheduler is currently being developed[15] with new features and fixes to the credit/priority management of the previous one. It tries to improve fairness, mainly the ability of a VM to get its "fair share" of CPU, defined in terms of weight, cap and reservation.

Another main target they are trying to achieve work well for latency-sensitive workloads. Ideally, if a latency-sensitive workload uses less than its fair share of CPU, it should run as well when the system is loaded as it does when the system is idle. If it would use more than its fair share, then the performance should degrade as gracefully as possible.

Another aspect it will improve are hyperthreads. A VCPU running on a core by itself will get a higher throughput than a VCPU sharing a core with another thread. Ideally, the scheduler should take this into account when determining the “fair share”.

Powering down cores or sockets into deeper sleep states can save power for relatively idle systems, while still allowing a system to perform at full capacity when needed. This new scheduler will eather implement this power-vs-performance tradeoff, or provide support for another system to do so.

Patches for its usage as well as some further information about its current development is available at the Xensource wiki web page [16].

Chapter 3

Load Balancing on Speed

Multicore processors and even several processor chips hardware computers are prevalent nowadays in many different systems. All hardware resources available within a physical system continues to rise at a very significant rate. It is very likely that the degree of the parallelism on-chip will significantly increase in the near future and processors will contain tens and even hundreds of cores. Making full use of this hardware and achieving a correspondent performance is nowadays a hard task. Virtualizing several different machines into a one-physical host is a common approach to make full use of this hardware. This virtualized machines may contain its own operating system with different types of applications running on them. Making a good scheduling and properly matching of the virtual machine needs to the physical resources is a key task to achieve performance and fairness. New parallel applications arise with different and wide targets. Very often these applications are focused in Scientific Computing. Allowing them to be executed in virtual machines without punishing performance is an important and challenging task.

It is proposed a generic technique to load balance parallel scientific applications written in SPMD style and executed within virtual machines. This technique is designed to perform well on multicore processors when any of the following conditions is met:

1. The number of tasks in an application might not be evenly divisible by the number of available cores;
2. Asymmetric systems where cores might run at different speeds
3. Non-dedicated environments where a parallel application's tasks are competing for cores with other tasks.

Virtual Environments, such as Xen, are written and optimized mainly for running Operating Systems with multiprogrammed commercial and end-user workloads. The Load Balancing on Speed[4] is a balancing technique designed to perform well on multicore processors in several and different situations:

1. The number of VCPUs assigned to an operating system might not be evenly divisible by the number of available physical cores
2. Asymmetric systems [17] where physical cores might run at different speeds
3. Non-dedicated environments where a parallel application running in a domain's virtual CPUs are competing for cores with other virtual CPUs for the same physical CPU

The developed scheduler manages all the VCPUs within the operating systems that are supposed to run parallel applications and uses migration to ensure that each VCPU is given a fair chance to run at the fastest speed available system wide.

3.1 Load Balancing

Load balancing is known to be extremely important in parallel systems and applications in order to achieve good performance. Balancing VCPUs in a virtualization environment like Xen could be similar to those applied to process withing an operating system. Scheduling methods from operating systems could be adapted and tuned in order to achieve a good performance in these virtualization environments.

The current design of load balancing mechanism present in Xen, mainly on credit scheduler makes some assumptions about the operating system's workload behavior. Interactive workloads are characterized by independent tasks that are quiescent for long periods of time (this is relative to CPU-intensive applications). Server Virtual machine workloads contain a large number of threads that are mostly independent and use synchronization for mutual exclusion on small shared data items and not for enforcing data or control dependence. To properly manage these workloads, load balancing algorithms in use do not start threads on new cores based on global system information.

Another implicit assumption is that applications are either single threaded or, when multi-threaded run in a dedicated environment, with properly selection and matching of VCPUs and physical CPUs. This is not always true, since we might want to use our virtual machine for different and heterogeneous environments. It is widely common to have different services and therefore different workloads running on the virtual machines withing a virtualization environment like Xen. Scheduler's load balancer must take this into account and balance properly VCPUs regarding their domain's contained workloads.

The main characteristics of actual existing load balancing implementations can be described as follow:

1. They are designed to perform best in the cases where in the cases where cores are frequently idle

2. Balancing uses a coarse-grained global optimality criterion (equal queue length using integer arithmetic).

These heuristics work relatively well for current commercial and end-user multi-programmed environments but are likely to fail for parallelism application characteristics is an essential step towards achieving efficiently utilization when virtual machines are running parallelism based workloads. This is also prone to fail in the task of achieving a good performance if the virtual environment is running over an asymmetric system. Developing scheduler support for proper virtual machine characteristics running parallel applications is an essential step towards achieving efficient utilization of highly heterogeneous virtualization environment where a wide variety of virtual machines with different services can be runned.

3.2 Load Balancing HPC applications

Most of the existing implementations of High Performance Computing on scientific applications use the SPMD programming model. The SPMD programming model which involves several phases of parallel computation followed by barrier synchronization. The OpenMP paradigm fully explored in this project provides SPMD parallel computation. This sort of computation model contravenes the assumptions made in the design of virtual system level load balancing mechanism: threads running within systems and therefore running in VCPUs are logically related, have inter-thread data and control dependence and have equally long life-spans.

SPMD applications commonly make static assumptions about the number of cores available and assume dedicated environments. This might be true in some scenarios with no virtualization and dedicated hardware, but it is certainly not true to the most cases of virtualization environment where no dedication is performed and a wide variety of different applications is executed within the virtual machines. When applications executed in virtual machines are SPMD programing model based, they are run with static parallelism or plainly restrictions on the degree of task parallelism due to the non existent parallel domain decomposition. For instance, many parallel jobs often request an even or perfect square number of processors, this is not properly balanced by the current operating systems, and therefore making the virtual machine have the required number VCPUs (even not matching the number of physical CPUs) should be properly balanced by the hypervisor's scheduler.

It is also important to take into account physical asymmetries or uneven VCPUs distributions across cores are likely to be founded in future systems. For instance, The Intel Nehalem processor provides the Turbo Boost technology [18] that, if the required situation is achieved, certain over-clocks cores until a temperature threshold is achieved. This result in cores running at different clock speeds. Some recently proposed OS designs such as Corey [19] or those under development at Cray and IBM provide for reserving cores to run only OS level services.

In order to achieve good performance for domains running SPMD applications requires that:

1. All tasks within the application make equal progress
2. The maximum level of hardware parallelism is exploited

Considering a two physical CPU system, with a total of three VCPUs within two running domains. One of the domains will have two VCPUs and the other just one. The Xen Scheduler will most likely assign two VCPUs to one of the physical cores and the other VCPU to the other physical core. This will lead to the domains running in the overloaded core perceiving the system as running at 50% speed. The impact on performance will be considerably. Administrators might be able to correct this behavior properly selecting the desired parameters of the scheduler in order to give priority to one of the domains, but full fairness and equality uses of physical CPU would be hard to achieve. The implemented approach also addresses this scenario by explicitly detecting and migrating VCPUs across physical CPUs run queues, even when the imbalance is caused by only one VCPU.

In dedicated environments where applications provide orders of magnitude performance improvements [20]. Most of the time virtualization environments are non-dedicates or oversubscribed. In this kind of systems, some form of yielding the processor is required for an overall progress. A thread that yields remains on the active run queue and hence the OS level load balancer counts it towards the queue length (or load). On the other hand, a VCPU that sleeps is removed from the active run queue, which enables the hypervisor's scheduler level load balancer to pull VCPUs onto the CPUS where VCPUs are sleeping.

3.3 Motivation For Speed Balancing

In SPMD applications threads have to synchronize on each barrier. When executed the parallel performance is that of the slowest thread and variation in the "execution speed" of any thread negatively affects the overall system utilization and performance. A particular VCPUs containing the parallel threads will run slower than the others due to running on the physical core with the longest queue length, sharing a core with other VCPUs with higher priority or running on a core with lower computational power (slower clock speed, like in an asymmetric system). In our implementation of load balancing on speed cores are classified in slow or fast cores depending on the "progress" perceived by the hypervisor. Figure 3.1 shows how the speed balancing algorithm would classify the cores in the previously mentioned example.
Consider N virtual CPUS from a domain running a parallel application with N threads. The operating system will see N CPUs and match each one of the appli-

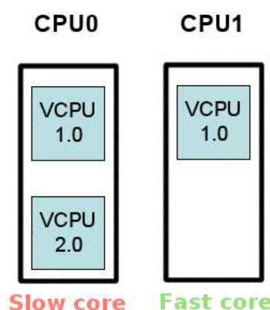

Figure 3.1: Classification of cores following Speed Balancing

cation threads to the CPUs. Let's considerer these N VCPUs are running in M physical homogeneous cores, $N > M$. Let V be the number of be the number of VCPUs per physical CPU $V = \lfloor \frac{N}{M} \rfloor$ Let FC denote the number of *fast* cores, each running V VCPUs and SC the number of *slow* cores with $V + 1$ VCPUs. Assume that VCPUs will execute for the same amount of time T secondsand balancing executes every B seconds. Intuitively, T captures the duration between two program barrier or sync points. With the Xen regular load balancing the total program running time under these assumptions is at most $(V1) * T$, the execution time on the slow cores.

We assume that migration cost is negligible, therefore the statements about performance improvements and average speed provide upper bounds. Assuming a small impact of VCPU migration is reasonable: The cost of manipulating VCPUs structures is small compared to a time slice, cache content is likely lost across context switches when threads (or in our case VCPUs) share cores. Li et al [21] use microbenchmarks to quantify the impact of cache locality loss when migrating tasks and indicate overheads ranging from μ seconds (in cache footprint) to 2 milliseconds (larger than cache footprint) on contemporary UMA Intel processors. For reference, a typical scheduling time quantum in Xen Hypervisor is 10ms.

Under this circumstances, using a fair per core scheduler the average VCPU speed is $\varphi * \frac{1}{V} + (1 - \varphi) * \frac{1}{V-1}$ being φ the fraction of time the VCPU has spent on a fast core. The Xen credit based balancing will not migrate VCPUs unless they go to sleep or block. This way the overall domain speed would be the one from the slowest VCPU $(V1) * T$. Instead, ideally, each VCPU should spend an equal fraction of time on the fast cores and on the slow cores (looking at the ideal for speed balancing to perform better than Xen, not the ideal for optimal load balance on every case). The asymptotic average VCPU speed becomes $\frac{1}{2*V} + \frac{1}{2*(V+1)}$ which amounts to a possible speedup of $\frac{1}{2*V}$.

Perfect fairness could be achieved but instead an argument is followed based on

necessary but not sufficient requirements: *In order for speed balancing to perform better than Xen provided balancing each thread has to execute at least once on a fast core.*

There are some constraints on the number of balancing operations required to satisfy this condition, indicating the threshold at which the speed balancer will be expected to provide a better performance. Below the mentioned threshold the two algorithms will be likely to perform in a similar manner. So when negative qualifiers are mentioned it is relating to achieving a similar performance than the Xen's actual default scheduler.

Lemma: *In order to satisfy the necessity constraint the number of balancing steps is:*

$$2 * \left\lceil \frac{SC}{FC} \right\rceil \tag{3.1}$$

In the case $FC < SC$, in each step of the algorithm one VCPU will be pulled from a slow queue into a fast queue. This simple action, migrating a VCPU will flip the core from slow to fast and the receiver from fast to slow, meaning that at each step it is given $FC * V$ VCPUs the opportunity of running in a fast core. This is done continuously for $\frac{SC}{FC}$ steps and at the end $N - SC$ VCPUs have run once on a fast queue. There are SC VCPUs left that had no chance at running fast. From all the queues containing these VCPUs we need to pull a different thread onto the available fast queue and run one interval without any migrations. This process will take $\frac{SC}{FC}$ steps. Then the total number of steps is $2 * \frac{SC}{FC}$.A similar reasoning shows that for $FC \geq SC$ two steps are needed.

This describes an actual algorithm for speed balancing which expected behavior can be discussed. Running queues run at most one thread per step and the rate of change is slow. Because of the fact that threads should not be migrated unless they have had a chance to run on a queue, balancing could start after $V + 1$ time slice: increasing load postpones the need for balancing. In the lemma it is provided the heuristics to dynamically adjust the balancing interval if an application behavior knowledge is available.

The whole domain running time is $Total_time = (V + 1) * T$ and according to the lemma the prerequisite for speed balancing to be profitable is the one shown in Equation 3.2

$$Total_time > 2 * \left\lceil \frac{SC}{FC} \right\rceil \; or \;\; (V + 1) * T > 2 * \frac{NmodM}{M - NmodM} \tag{3.2}$$

Assuming that the balancing algorithm implemented is performed synchronously by all cores. Note that the implementation performed in this project uses a distributed algorithm so VCPU migrations might happen with a higher frequency, proportional to the number of cores.

With this argument homogeneous systems can be easily extended to heterogeneous systems where cores have different performance by weighting the number of threads per core with the relative core speed.

3.4 Algorithm

It has been provided a balancer that manages the domain VCPUs on user requested domain. Domains can be *marked* as highly parallel and those are the one that will be balance in order to achieve the performance benefits. The implementation does not require any application modifications, it is completely transparent to the domains users and the Xen system administrator. Also, it does not make any assumptions about implementation details, specially no assumptions about thread *idleness* are made (this is an application-specific notion, or about synchronization mechanism). Also no assumptions on synchronization mechanisms are made: busy-wait, spin locks, or mutexes. Blocking or sleep operations are captured by the algorithm since they will be reflected by increases in the speed of their core co-runners.

For the implementation of the algorithm the speed of a VCPU would be defined as $speed = \frac{t_{running}}{t_{runnable}}$, Where $t_{running}$ is the time the VCPU has been running on the core and the $t_{runnable}$ is the time the VCPU has been runnable This is an improvement over previously proposed Load Balancing on Speed algorithms [4], since we believe this is a much more better way of capturing the share of CPU time received by a VCPU. This would also be easily adapted to capture behavior in asymmetric systems. It is also simpler than using the inverse of the queue length as a speed indicator mainly because that would require weighting VCPUs by their priorities, which can have different effects on running time depending on the task mix and the associated scheduling classes. This is consider to be a more elegant approach than the one proposed by Hofmeyr et al. [4]. This balancer measured speed like $speed = \frac{t_{exec}}{t_{real}}$ which is not like to achieve the same amount of performance as ours in the sequential parts of SPMD applications. Taking into account the runnable AMD running time of the VCPU would make our algorithm aware of the sequential parts of the executing application leading to a better overall performance. Some other way of measuring speed will be discussed further in the conclusions section, Section 5.

Our implementation of the speed balancing uses a timer on each physical CPU structure in the Xen's scheduler. Our approach is made scalable and distributed along the CPUs. The timer of each CPU will tick our balancing function peri-

odically, making our function start running. The balance function will check for imbalances, correct them pulling VCPUs from a slower core to the local core (if possible) and then sleep again. The frequency which the balancer is ticked (balance interval) determines the frequency of migrations. This is a parameter that should be tunned properly because it is likely possible that can impact the performance of the balancer. From previous research we have taken the optimum interval to be 100 ms, although further values will be discussed in section 5.

Note that in the following description the notion of a core's speed is an entirely function specific notion. When several applications run concurrently each might perceive the same core differently based on the task mix. When our balancer is ticked, it performs the following steps:

1. For every VCPU V_i in the current physical CPU C_j it obtains the speed VS_ij over the elapsed balance interval, using the $t_running$ and the $t_runnable$ metrics associated to each VCPU.
2. It computes the local core speed CS_j over the balance interval as the average of the speeds of all the threads on the local core: $CS_j = average\left(VS_j^i\right)$
3. It computes the global core speed S_{global} as the average speed over all cores: $S_{global} = average\,(CS_j)$.
4. It attempts to balance if the local core speed is greater than the global core speed: $S_j > S_{global}$

The step 4 says the balancer attempts to balance, this means that it will search for a suitable remote core C_k to pull threads from. A remote core C_k will be suitable if its speed is less than the global speed ($S_k < S_{global}$) and it has not been involved in migration in the last two balance intervals. This block post-migration is two balance intervals because we need to ensure that the balanced threads on both cores have run for a full balance interval and the core speed values are not stale. This heuristic has the side effect that it allows cache hot threads to run repeatedly on the same core. Once it finds a suitable core C_k , the balancer pulls a thread from the remote core C_k to the local core C_j. The balancer chooses to pull the thread that has migrated the least in order to avoid creating tasks that migrate repeatedly.

After balance have been performed the timer will get set up to wake up in the next 100 milliseconds, in order to perform a new balance. This occurs for each thread and each 100 ms.

3.5 Speed Balancing Implementation on Xen

To implement the Load Balancing on Speed a new Xen Scheduler have been developed. In our code a file *sched_speed.c* contains the code of our scheduler. The

scheduling of the core queues has been implemented following the Credit Scheduler algorithms which seemed to perform well and are highly configurable from an administrator point of view, but any other matching approaches may be used to test our balancing algorithm [22].

Several additions have been made to the structures usually managed by a scheduler. Using the structures reserved pointer (as showed in Section 2.1) we have stored the needed info, of special mention are: the speed values, the number of balance intervals since last migration, timers for calling the balance function and lockers to prevent from problems of atomicity.

During the initialization of the physical CPU structure along with all needed variables of the timer is set up to call the balancer. Later when the Xen's SMP subsystem is alive the CPU timers are kicked to call the balance function in the next 100 milliseconds.

When the balance function is called it will perform the steps explained in Section 3.4. If those steps required so, the migration function will be called to perform a balance action between the physical CPUs selected, also variables will be updated properly.

Finally, it will tell the timer to wake the function up again in the next 100 milliseconds. The balance interval can be easily changed by a parameter within the system, in order to be able to perform different tests.

3.5.1 Algorithm and Structures

These are some of the main structures implemented in order to make our scheduler work. They are similar to those on the credit scheduler since our version of the algorithm maintains the same behavior per queue as the Credit Scheduler. As commented the big difference comes with the load balancer.

Listing 3.1 shows the constant that defines the size of the time slice within the balancers from the cores wake up and start collecting the information in an attempt to balance cores. It is a constant because is an important parameter that we might want to tune in order to look for the best performance of the algorithm.:

```
1 #define LBS_MSECS_PER_BALANCE 100
```

Listing 3.1: Balancing interval constant

For the purpose of information collecting some macros have been coded. Macros in listing 3.2 access the proper Xen's structures in order to obtain the current running

and runnable time that are used in for VCPU's speed calculation in our approach of the algorithm, as deeply explained on Section 3.4.

```
1 #define t_running(_lbs_sched_vcpu)     _lbs_sched_vcpu->vcpu->runstate.
      time[0]
2 #define t_runnable(_lbs_sched_vcpu)    _lbs_sched_vcpu->vcpu->runstate.
      time[1]
```

Listing 3.2: Running and runnable time macros

As fully explained in Section 2.1, making use of the special pointers implemented in Xen, reserved for the schedulers, we have defined some structures for our schedulers. This structures helps us to keep track of the system in order to make the proper scheduling and balancing decisions.

Listing 3.3 shows our scheduler's physical CPU structure. This represents a core from the systems and holds its related information. For balancing purposes the most important variables are: *cpu_speed*, *last_suc_balance*, *numpulled* and *numpulled*. The last two were added for debug and statistics collection, specially those statistics shown in graphs on Section 4.2.4. The *cpu_speed* as it might been guessed holds the speed of the CPU. And finally *last_suc_balance* helps us to keep track of the last successful balance in order to implement the rule of *"it has not been involved in migration in the last two balance intervals"*, fully explained on Section 3.4

```
 1
 2 struct lbs_sched_pcpu
 3 {
 4       struct list_head runq;
 5       uint32_t runq_sort_last;
 6       struct timer ticker;
 7       struct timer tticker;
 8       struct timer ticker_LBS;
 9       unsigned int tick;
10       unsigned int idle_bias;
11       uint64_t cpu_speed;
12       uint16_t last_suc_balance;
13       int numpulled;
14       int numpushed;
15 };
```

Listing 3.3: LBS Scheduler physical CPU structure

Code within Listing 3.4 the implementation of the VCPU structure for our scheduler. It holds the relevant information for VCPUs in order to be properly balanced and scheduled. Of special mention is the *speed* variable which holds the speed value associated with the VCPU. This is calculated making use of the macros previously explained in this Section and the algorithm on Section 3.4,

```
1
2  struct lbs_sched_vcpu
3  {
4          struct list_head runq_elem;
5          struct list_head active_vcpu_elem;
6          struct lbs_sched_dom *sdom;
7          struct vcpu *vcpu;
8          atomic_t credit;
9          s_time_t start_time;
10         uint16_t flags;
11         int16_t pri;
12         uint64_t speed;
13 };
```

Listing 3.4: LBS Scheduler VCPU structure

There is also a domain structure that holds information regarding domains, see Listing 3.5. There are some variables important for scheduling. The only balancing used variable is the last one, *hpcdomain*, which indicates to our algorithm if a domain is intended for parallel computing and should be balanced on speed.

```
1  struct lbs_sched_dom
2  {
3          struct list_head active_vcpu;
4          struct list_head active_sdom_elem;
5          struct domain *dom;
6          uint16_t active_vcpu_count;
7          uint16_t weight;
8          uint16_t cap;
9          uint16 hpcdomain;
10 };
```

Listing 3.5: LBS scheduler domain structure

The scheduler private variables are implemented as shown in Listing 3.6. The two more relevant for our balancing algorithm are the *global_ speed* and the *numbalances*. The first one helps us to keep track of the global speed, average of all the CPUs within the system, as explained on Section 3.4. The second one is mostly for statistics purposes and tracking of the algorithm.

```
1  struct lbs_sched_private
2  {
3          spinlock_t lock;
4          struct list_head active_sdom;
5          uint32_t ncpus;
6          struct timer  master_ticker;
7          unsigned int master;
8          cpumask_t idlers;
```

```
 9       uint32_t weight;
10       uint32_t credit;
11       int credit_balance;
12       uint32_t runq_sort;
13       uint64_t global_speed;
14       int numbalances;
15   };
```

Listing 3.6: Scheduler private variables structure

Appendix A includes the main function of our scheduler. For clarity and easy reuse of the code it is strictly implemented following the defined rules shown in Section 3.4. The four balancing steps mentioned above are clearly defined and implemented. Comments are also added to make it easy to understand and to identify them.

Chapter 4

Experiments and results

In order to prove our assumptions about Load Balancing on Speed, the algorithm has been implemented on the last available version of the Xen hypervisor, 4.0.1. Then tested using several experimental environments involving virtual machines and applications running at the same time. In this section we present the most representative of the obtained results in order to be able to achieve some interesting conclusions.

4.1 Experimental Setup

The load balancing on Speed algorithm has been implemented as a new Scheduler to Xen. We have used the latest resleased Xen hypervisor package from the Debian project [23], from the unstable (squeeze) distribution. The kernel used is also the latest in this GNU/Linux distribution, the Linux 2.6.32-5, compiled for use with the Xen hypervisor. Making use of the debian package management tools, sources have been modified, compiled, repacked and installed on the target testing host.

A debian testing (squeeze) distribution along with unstable version of the modified Xen-hypervisor and the Xen-tools packages (for proper VM management) have been installed on the test system. Table 4.1 shows the physical configuration of the system used for testing.

Processor	Xeon X5670 2 chips x 6 cores (2,93 GHz)	
	L1 Cache (per Core)	32KB
	L2 Unified Cache	256KB
	L3 Unified Cache	12MB
Memory	48 GB 3xDDR3-1333	
Operative System	GNU/Linux 2.6.32-5-amd64	
G++ Compiler 4.4.5	-O3 -fopenmp	

Table 4.1: Intel Westmere architecture used in the experiments

The Intel Westmere constitutes the new 32nm processing technology from Intel but it maintains all the aspects of the Nehalem microarchitecture, like the previously mentioned Turbo Boost.

For debugging purposes and initial development, the IPMI tool and its Feature Serial Over Lan (SOL) have been used. With the proper set up and the required tools this technology allows us to dump the serial port output through the LAN port. This way we can see the serial output from almost any computer with an internet connection. So it has been possible to debug the scheduler and the Xen boot up process from a different machine through the SOL technology, compiling the hypervisor with the proper debug flags. This is really useful to debug the hypervisor system during the boot up process and when hangs occur. From the serial console the Xen debug keys can be used in order to perform several intersting debugging actions such as dumps of different structures, reboots or memory info. When Xen is properly booted xentrace also becomes an intersting debugging tool to see what is really going on in the system, although the use of the xenalyze tool is needed to make of the output of this tool human readable.

4.1.1 Virtualization environment setup

As discussed before, a virtual environment holds a wide variety of virtual machines with different applications. In order to test the algorithm we have come up with our own virtualization environment. We have assume a scenario where many different applications are executed every day, the typical workload from many HPC research groups [24], with almost all the research and services machines being multicore.

It is common for the users to have the need of a web pages or a web services that needs to be allocated somewhere. The system administrators would have multicore machines with a virtualization environment set up [25]. This way they can make full use of the multicore machine and at the same time serve several different needs with just one physical host. After setting up the machine, another request for a

computer to execute parallel application comes. It is allocated according to the best the administrators can do to fit the user's needs.

To address these needs we are assuming the virtual environment described in Table 4.2.

Virtual Machine	Parallel Application Virtual Machine **Domain 1**										Java Web Server Virtual Machine **Domain 2**			
VCPUs	0	1	2	3	4	5	6	7	8	9	0	1	2	3
Mapped CPUs	0	1	2	3	4	5	6	7	8	9	8	9	10	11

Table 4.2: Virtualization Environment used for the tests

As you can see, the cores numbered 8 and 9 are shared between the two virtual machines. The reason for doing such a thing is because the Java server is not likely to use the 4 VCPUs but it might use them during a peak of intensive workload. Then, if a parallel application is running and the Java server in Domain 2 is making full use of the 4 VCPUs it is desired for the parallel application to be able to still achieve a good performance.

For the Java server we have chosen the SPECjbb2005 [26] suite which is a set of SPEC's benchmarks in order to evaluate the environment with the performance of the Java side of a server. It emulates a there-tier client/server system (emphasizing the middle tier), considered the most common type of server-side Java application today. The benchmark exercises the implementations of the JVM (Java Virtual Machine), JIT (Just-In-Time) compiler, garbage collection, threads and some aspects of the operating system. On the side for Domain 1 the SPEC omp2001 [27] have been used in order to test a parallel application. This is a SPEC benchmark suite for evaluating performance based on OpenMP applications. Table 4.3 shows the benchmarks from the whole suite selected for the tests.

Categories	Benchmarks
Highly parallel - CPU Intensive	fma3d, ammp, wupwise
Highly parallel - Memory Intensive	art, equake, applu, swim

Table 4.3: SPEC omp2001 selected benchmarks

Our Load Balancing on Speed scheduler will just balance VCPUs from Domain 1, the virtual machine with running the parallel application (previously selected by the administrator). This will make the balancer migrate only the VCPUs from Domain

1 and just within the CPUs assigned to that Domain. As expected, and discussed in the next section, not only Domain 1 will be affected by the balancing but also the performance of other virtual machines, those sharing cores with Domain 1, will change.

4.2 Performance results

When threads reach a barrier they often start spinning until the defined threshold is obtained. The idea is that waiting on the spin lock, the thread will be repeatedly polling a value, while is queued to enter a critical section. The purpose of spinning is that the desired synchronization barrier may be released soon enough that it is worth the wait instead of incurring the overhead of yielding the processor o blocking until the lock is released

Here we present and discuss the results obtained from the different runs conducted within the virtualization environment. We have performed some basic runs an then we have obtained the results for different given values of the spin parameter, indicated to the gcc compiler by an environment variable. The initial idea is to avoid running spinning threads on *fast* cores and save these "power hungry" cores for other threads.

4.2.1 Baseline comparision

Un this setup, the two virtual machines are up and running. The Java server is executing its Java instances as usual and we measure the values of the SPEC omp2001 applications with the default scheduler of Xen and our implementation of the Load Balancing on Speed.

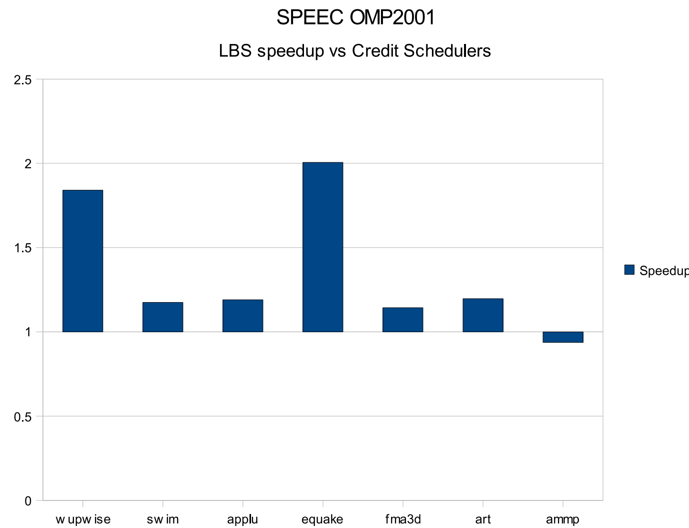

Figure 4.1: Speedup of a basic run of the set up

The speed up for our load balancer is shown in the Figure 4.1 As you can see our LBS scheduler seems to perform better than Xen's default one. This measures are just for the Parallel application's Virtual Machine which actually is the one we are trying to balance. The performance is better because the Credit Scheduler from Xen will assign VCPUs to the shared cores. These cores are shared with the other virtual machine, they will probably be *slow*. Having the same VCPUs running on the *slow* cores would hit performance of the parallel application.

Our balancer will detect these *slow* cores and pull VCPUs from them in order to make all the Domain's VCPUs run the same amount in *slow* than in *fast* cores. This then translates to a better overall performance and the correspondent speedup shown in Figure4.1

Our original idea was that speedup would be much higher for the CPU intensive benchmarks since migrations deteriorate the performance of memory intensive applications. However, some benchmarks such as ammp, we do not observer this behavior. Up to know we are not able to propose a convincing theory about this. A future deeply debug of the execution of these benchmarks could determine which factors decrease our speedup.

4.2.2 Performance impact due to sharing CPUs

We have tried to measure the impact of sharing the CPUs within the two studied schedulers. First we have measured the SPEC omp2001 selected benchmarks when their Domain is the only one in the system. Then the same applications have been tested with the full scenario up (see Table 4.2), and running the SPEjbb205 on

Domain 2.

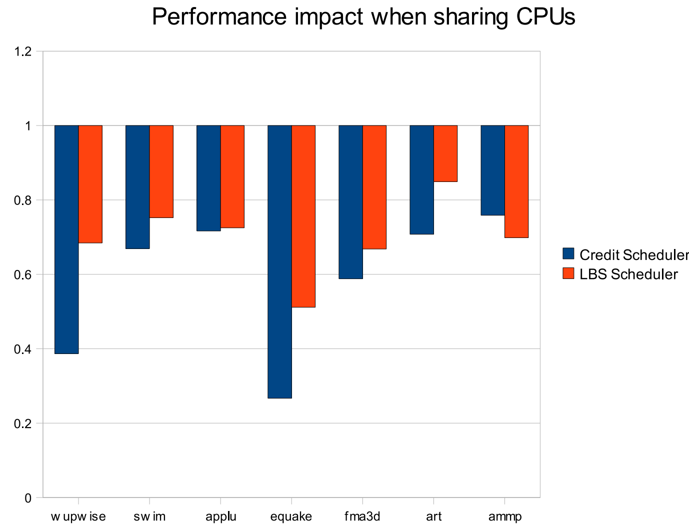

Figure 4.2: Peformance impact in the schedulers due to cpu sharing

Figure 4.2 shows the results of the measured scenario. There is an understandable performance punishment due to the sharing of the 8th and 9th CPUs with the other virtual machine. The results show, again how our LBS Scheduler is able to better manage CPU shared environemts than the Credit Scheduler. Because of the implemented algorithm, LBS Scheduler will balance the VCPUs assigned to the *slow* CPUs making the threads progress "toghether" in an effort to make then achieve the synchronization barriers all at the same time. On the other hand the Credit Scheduler will not be aware of this shared *slow* cores leading to a bigger impact on performance.

4.2.3 Adjusting the spin threshold

As explained earlier in this Section, the spin threshold is the number of instructions that the thread will be wating in the spin lock. Spining means the thread will be repeatedly pollin a value. This is a threshold that is likely to impact in the performance of our balancer, since keeping the threads "running" after reaching the lock will make them still use CPU power.

Several values have been choosen so we can have an overview of the real impact of this threshold in the studied schedulers, specially on our proposed algorithm. Four representative values have been selected for the test: 0, 100, 1000 (1K) and 20M.

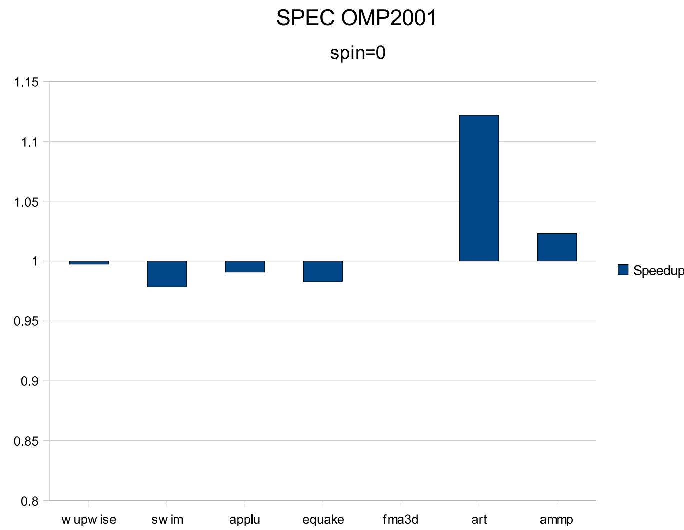

Figure 4.3: LBS speedup for a spin threshold of 0

Figure 4.3 shows the case with no spin at all (spin=0). There is not much speedup achieved for the Parallel Application domain using LBS Scheduler. That is because when no spin is used, the threads go to sleep when they are done and have reached the synchronization barrier, leaving the VCPU with no workload that, in consecuence, might also go to sleep. Credit Scheduler will detect this situation and take advantage of the *free* cores, not getting his performance highly impacted by the fast slowest threads.

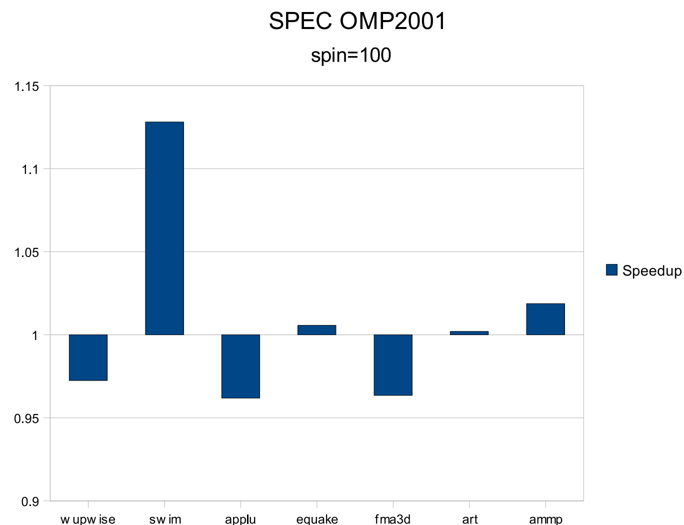

Figure 4.4: LBS speedup for a spin threshold of 100

In Figure 4.4 results for the spin value of 100 are shown. The speedup is slightly better than in Figure 4.3 (with no spin). This is because the threads will spin for a little before going to sleep an that seems to benefit the Load Balancing on Speed Scheduler.

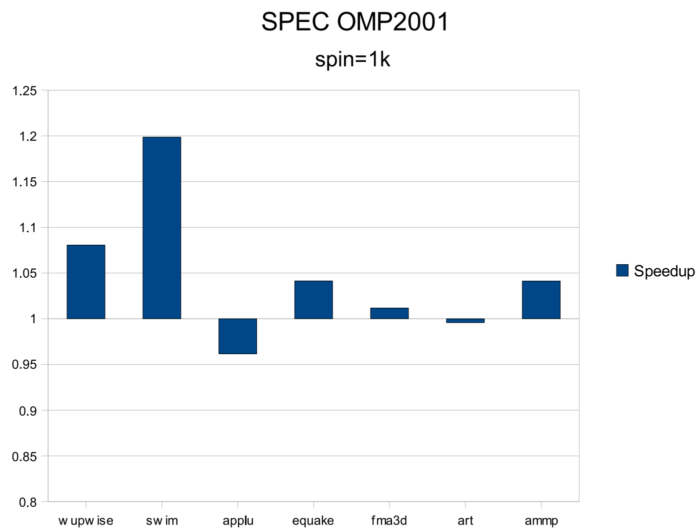

Figure 4.5: Speed up of a basic run of the set up

The values of speedup when there is a spin threshold of 1k instructions are shown in Figure 4.5. This threads will be spining for 1k instructions and this, although is not uniform, seems to perform better. Speedup is achieved in almost all threads so 1k spin value is starting to be a profitable value in order for our LBS Scheduler to achieve a good performance against the default Credit Scheduler.

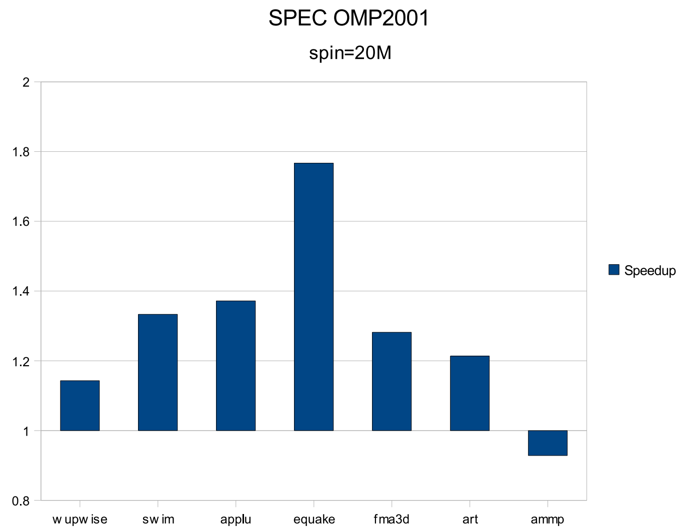

Figure 4.6: Speedup of a basic run of the set up

The top speedup values achieved tunning the spin threshold is the one for the 20M. Figure 4.6 shows these results. The credit scheduler will not ve able to get noticed of the threads that achived the synchronization barrier at least for 20M instructions, since they will be spining until then. On the other hand, our Scheduler will take this VCPUs and balance them properly according to the CPU speeds.

From the obtained results we can tell that, there is a tendency for our scheduler to perform better the greater the spin value is. At least this is true for specs like *wupwise*, *swim*, *equake*... but what is undeniable is that almost all the specs get a good speed up with our scheduler with the spin value of 20.000. A better overview of all these results is provied in Figure 4.7, which includes all speedups obtained for all the benchmarks and the different spin threshold values. From there you can tell that speedup is likely to increase along with the spin threshold, probably until is equal or higher than the biggest wating a thread will have on a synchronization barrier in the benchmark executed. Point after what increasing the spin value will make no performance difference since threads will spin the same amount of time (which is all the time they are waiting on the lock).

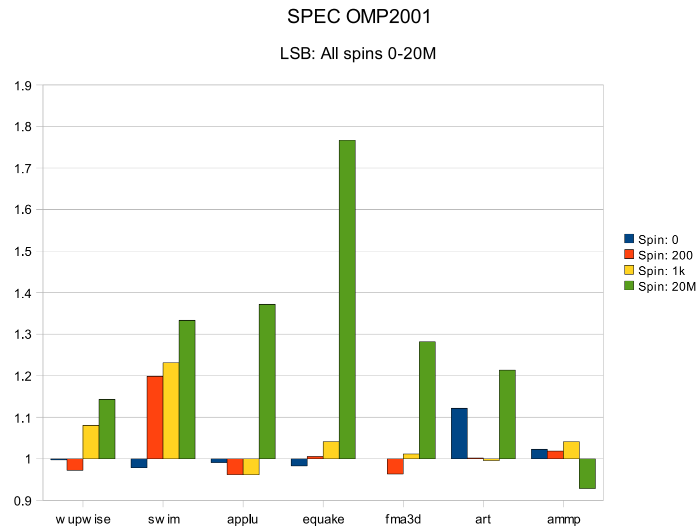

Figure 4.7: Java server benchmark perform in a regular run for setup

4.2.4 SPEC jbb 2005

It is also important not to impact the performance of the other virtual machines so we have taken track of the behavior of the Virtual Machine containing the Java Server. Not only the performance is not impacted by our scheduler but improved. Figure 4.8 shows the performance improvement obtained with our scheduler of the execution of the SPEC jbb 2005 when executing parallel applications (the SPEC omp2001) on the other virtual machine.

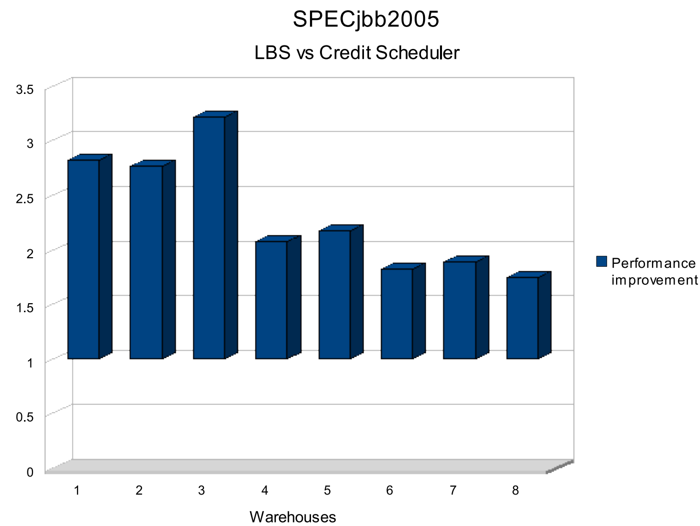

Figure 4.8: Java server benchmark perform in a regular run for setup

This unexpected results are reasonable taking into account that the percentage of shared VCPUs by this virtual machine is very high (50%). We are balancing the other virtual machine but at the same time speeding up this one. The Balancer is selecting the threads from the slowest cores, and the candidates for that are the shared cores, which will probably be the *slowest* in the system. Then VCPUs from the HPC virtual machine are pulled to another *faster* cores available for this virtual machine, leaving VCPUS of the Java Server alone in the core. This leads to the VCPUs from the Java Server spending more time alone in the shared cores than with the Credit Scheduler Finally that results in a considerably speed up of the non-balanced virtual machine.

In order to prove our hypothesis we have collect some statistic, debugging the Xen Hypervisor, of one of the benchmark's execution. Figure 4.9 shows that, with a big difference, the main two cores where cores are pulled out are the two shared ones: 8 and 9. They also are the ones where never receive cores from other CPUs.

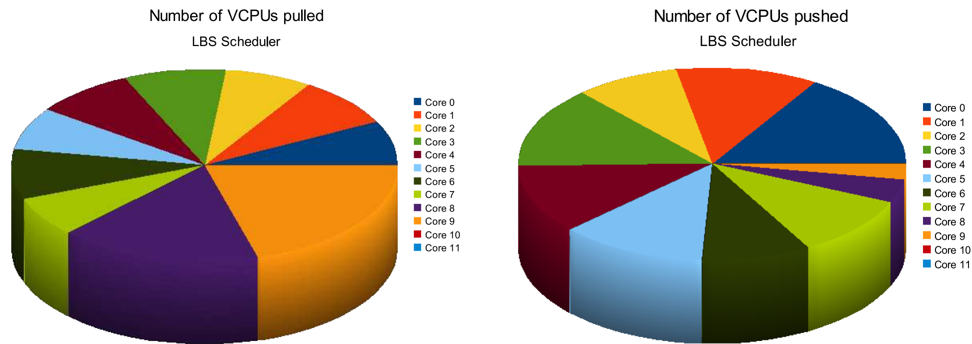

Figure 4.9: Number of VCPUs being pulled and pushed per core

This is translated to most of the time our Balancing algorithm leaving cores 8 and 9 free for the VCPUs of the second Domain. Therefore, increasing the 2nd Domain's performance. Figure 4.10 show the relation between pulled and pushed VCPUs and almost all cores are been pushed more cores than pulled but the two shared ones.

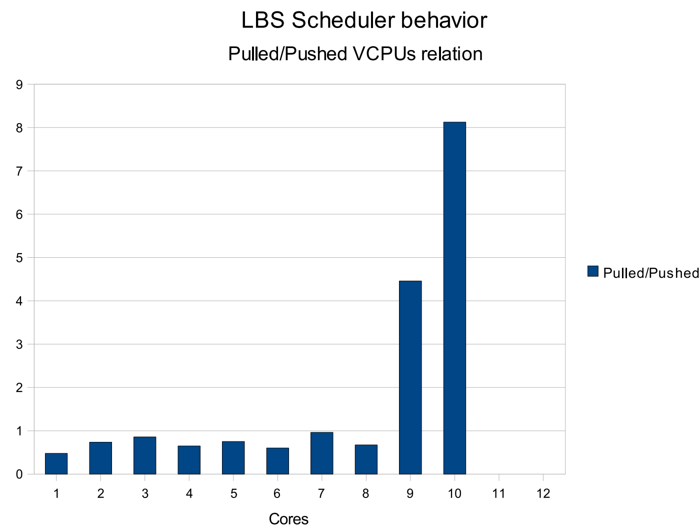

Figure 4.10: Ratio of Pulled/pushed VCPUS per core

Since this difference between the default scheduler and LBS is attributed to the high percentage of sharing CPUs of this machine we have finally perform some runs in order to see the evolution of speedups changing the number of shared CPUs.

4.2.5 Increasing number of shared cores

In order to prove the assumptions made on previous section we have increased and compared the number of shared cores.

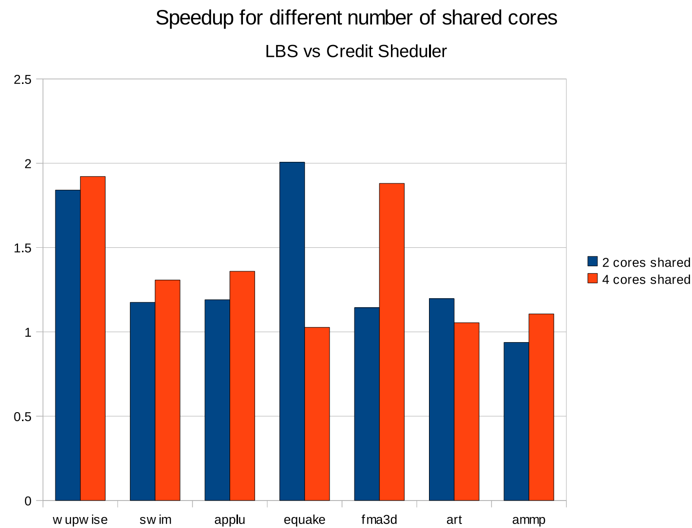

Figure 4.11: Speedup for 2 and 4 shared cores

We have given 6 cores to Domain 2 and mapped these two new cores to other two shared CPUs with Domain 1. This way we are increasing the competition for CPU time up to 4 cores instead of 2, which is the previous case. The results are shown in Figure 4.11. The speedup against the Credit Scheduler for 4 shared cores is bigger than it is for 2. That is because the balancing of our algorithm performs great in the task of mitigating the effects of this *slow* shared cores, or, at least, better than Xen default's scheduler. In consequence we can conclude that when scenarios have a high number of shared cores, our algorithm is likely to perform better than the Xen's default option.

Chapter 5

Conclusions and Future Work

Actual and future machines are likely to be heavily multicored. These systems are able to achieve such a high performance that a single user or a single operating system will probably not be able to get the most out of it. An approach to make full use of this potential is virtualization. Scheduling and therefore properly balancing is a key step in achieving this full use of the hardware's potential.

Here he have proposed the Load Balancing on Speed algorithm for use within the Xen virtualization system in order to accelerate the virtual machines dedicated to intensive execution of parallel applications.

In an effort to reproduce the most common virtualization scenarios we presented an heterogeneous one where parallel execution applications coexist with web server services. Results have shown that the number of instructions a thread spins is relevant in order to achieve the best performance. Also, that not only our balancing algorithm will be beneficial to our parallel application's virtual machine but to all the other virtual machines that share CPUs with it. Finally, we have seen that our algorithm achieves higher improvements as we increase the number of shared cores betwen the virtual machines. So the more crowded our virtualization environment is, the higher speedup is our algorithm likely to achieve.

Some improvements and tunes can still be applied to our algorithm in order to improve it and guess when is better to make use of it. Next section will further discuss this along with some related work.

5.1 Future Work

Without global synchronization, our algorithm cannot guarantee that each migration will be the best possible one. Each balancer makes its own decision, independent of the other balancers, which can result in a migration from the slowest core to a core that is faster than average, but not actually the fastest core. To help break cycles

where tasks move repeatedly between two queues and to distribute migrations across queues more uniformly, it might be interesting to try to introduce randomness in the balancing interval on each core. Consequently the elapsed time since the last migration event will vary randomly from one thread to the next, from one core to the next, and from one check to the next. If knowledge about application characteristics is available, the balancing interval might be further tuned.

Our formula for measuring the speed is an approximation to the "real" speed of a VCPU. To obtain this real speed a track of the progress of the program is needed. Only knowing the progress made by a program in a certain amount of time will give us the "real" speed of the thread. In order to do that in the Xen hypervisor system some hypercalls might be need to be inserted into the guest domains' operating system but a more accurate measure and therefore performance is expected.

Several other tunes and improvements to our algorithm might be done. Balancing the whole system could achieve overall performance increase for certain scenarios, or looking for the best scenario to perform the algorithm are some of the still available works to do.

Properly scheduling VCPUs is not an easy task to do, it has to achieve some goals, like fairness, performance, flexibility... As discussed in Section 2 the VCPU scheduling present in Xen is pretty similar to the way an operating system likely GNU/Linux will schedule the process. This leaves a wide open door for trying to port ideas from different operating systems schedulers to virtualization environments like Xen.

Appendix A

The Load Balancing on Speed Pseudocode

```
1
2 static void
3 balancing_on_speed (void *_cpu)
4 {
5     int cpu = (unsigned long)_cpu;
6     struct lbs_sched_pcpu *spc = LBS_SCHED_PCPU(cpu);
7     struct list_head *iter;
8     uint64_t total_speed_pcpu = 0U;
9     uint16_t num_vcpus = 0U;
10    uint16_t num_cpus = 0U;
11    uint64_t global_sum_speed = 0U;
12    int i;
13    int cpus_visited = 0;
14
15     //Reset speeds
16     spc->cpu_speed=0;
17     lbs_sched_priv.global_speed = 0;
18
19     //Inc the last balance counter
20     spc->last_suc_balance += 1;
21
22
23    //1.- For every VCPU on the local pcpu compute the speed
24    //(should be over the elapsed interval)
25    list_for_each( iter, RUNQ(cpu))
26    {
27        struct lbs_sched_vcpu *iter_svc = __runq_elem(iter);
28
29        if (t_running(iter_svc)!=0 && t_runnable(iter_svc)!=0)
30            iter_svc->speed=muldiv64(t_running(iter_svc),
31                                     (uint32_t)1000,
32             (uint32_t)t_runnable(iter_svc));
33
34        total_speed_pcpu += iter_svc->speed;
```

```
35            num_vcpus += 1;
36
37        }
38
39        //2.- Compute the local core speed
40        if (total_speed_pcpu!=0 && num_vcpus!=0)
41        {
42            spc->cpu_speed = muldiv64(total_speed_pcpu, 1,
43                                      (uint32_t)num_vcpus);
44        }
45
46
47        //3.- Compute the global speed
48        for_each_online_cpu ( i )
49        {
50            spin_lock(per_cpu(schedule_data, i).schedule_lock);
51            spc = LBS_SCHED_PCPU(i);
52            if(spc->cpu_speed!=0)
53            {
54                global_sum_speed += spc->cpu_speed;
55                num_cpus += 1;
56            }
57            cpus_visited += 1;
58            spin_unlock(per_cpu(schedule_data, i).schedule_lock);
59        }
60
61        if (global_sum_speed!=0 && num_cpus!= 0)
62            lbs_sched_priv.global_speed = muldiv64(global_sum_speed,
63                                                    1, (uint32_t)num_cpus);
64
65        //4.- It attempts to balance if the local core speed
66        //    is greater than the global core speed
67        spc=LBS_SCHED_PCPU(cpu);
68        if (spc->cpu_speed > lbs_sched_priv.global_speed)
69            balance(cpu);
70
71        //Setting the timer to wake in the next 100 ms
72        set_timer(&spc->ticker_LBS, NOW()
73                  + MILLISECS(LBS_MSECS_PER_BALANCE));
74    }
```

Bibliography

[1] Gnu general public license v3. `http://www.gnu.org/licenses/gpl.html`.

[2] The bsd license. `http://opensource.org/licenses/bsd-license.php`.

[3] Gnu free documentation licnese. `http://www.gnu.org/copyleft/fdl.html`.

[4] Steven Hofmeyr, Costin Iancu, and Filip Blagojević. Load balancing on speed. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–158, New York, NY, USA, 2010. ACM.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] David Chisnall. The definitive guide to the xen hypervisor, 2007.

[7] AMD Inc. Advanced Micro Devices. Amd virtualization codenamed "pacifica" technology, secure virtual machine architecture reference manual. May 2005.

[8] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[9] Steven Hand, Andrew Warfield, and Keir Fraser. Hardware virtualization with xen. *USENIX ,login magazine 32(1)*, 2007.

[10] Rodrigo S. Couto, Hugo E. T. Carvalho, Lyno Henrique, G. Ferraz, Miguel Elias, M. Campista, Luís Henrique M. K. Costa, and Otto Carlos M. B. Duarte. 1 cpu allocation on xen virtual networks.

[11] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[12] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[13] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, 1999.

[14] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications, 1997.

[15] George W. Dunlap. Scheduler development update. *Citrix Systems R.D. Ltd, UK*, 2010.

[16] Xensource wiki page. `http://wiki.xensource.com`.

[17] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[18] Intel turbo boost technolgy. `http://www.intel.com/technology/turboboost/`.

[19] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. 8th USENIX Symposium on Operating Systems Design and Implementation, In Proceedings of the (OSDI '08), 2008.

[20] R. Nishtala and K. Yelick. Optimizing collective communication on multicores. First USENIX Workshop on Hot Topics in Parallelism (HotPar'09), 2009.

[21] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 65–74, New York, NY, USA, 2009. ACM.

[22] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 97–108, New York, NY, USA, 2010. ACM.

[23] Debian project. `http://www.debian.org`.

[24] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[25] William von Hagen. Professional xen virtualization, 2008.

[26] Spec jbb205 benchmarks. http://www.spec.org/jbb2005/.

[27] Spec omp2001 benchmarks. http://www.spec.org/omp/.

List of Figures

1.1 Ring Usage in native and paravirtualized systems 8
1.2 System calls in native and paravirtualized systems 9
1.3 Xen Architecture Overview. Illustration from Novell.com 11
1.4 A network package sent from an unprivileged domain 12

2.1 Schedulers behavior according to their weight 19
2.2 Example of a system running with a Credit Scheduler 24

3.1 Classification of cores following Speed Balancing 31

4.1 Speedup of a basic run of the set up 43
4.2 Peformance impact in the schedulers due to cpu sharing 44
4.3 LBS speedup for a spin threshold of 0 45
4.4 LBS speedup for a spin threshold of 100 45
4.5 Speed up of a basic run of the set up 46
4.6 Speedup of a basic run of the set up 47
4.7 Java server benchmark perform in a regular run for setup 48
4.8 Java server benchmark perform in a regular run for setup 49
4.9 Number of VCPUs being pulled and pushed per core 50
4.10 Ratio of Pulled/pushed VCPUS per core 50
4.11 Speedup for 2 and 4 shared cores 51