TECHNICAL UNIVERSITY DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE FOR SYSTEM ARCHITECTURE
CHAIR FOR OPERATING SYSTEMS
PROF. DR. HERMANN HÄRTIG

# Großer Beleg

# Scheduling Operating-Systems

Stephan Diestelhorst
(Mat.-No.: 2854416)

Tutor: Dr. Ian Pratt, Dipl.-Inf. Jean Wolter

Dresden, June 5, 2007

# Contents

## Abstract

Virtualisation has been an *en vogue* topic for systems' engineers in the last decade, with a multitude of open-source solutions emerging. While being used in commercial systems for much longer, commodity hardware has just recently become capable enough to make this approach feasible on small- to mid-scale servers, workstations and even laptop computers.

But as available computing power still continues to rise, simultaneous virtualisation of many guest systems on a single machine becomes feasible. Together with the trend towards thread-level parallelism, multi-thread, multi-core and multi-processor architectures, the question of allocating CPU-time to the guests becomes crucial for overall system's performance and accounting in commercial scenarios.

This paper presents SEDF, a multi-purpose scheduler for Xen, an open-source virtual machine monitor, including support for firm guarantees, bandwidth reservations and parallel architectures, such as multi-cores, SMPs and SMTs.

## Acknowledgments

The present paper would not have been possible without contributions of all kinds by many people, to whom I express my deepest gratitude and sincere apologies, if you slipped my list!

Most of the research and programming of the content of this paper was done at the University of Cambridge (UK), at the System's Research Group in the Computer Laboratory and was funded by the DAAD's "Excellence Programme".

Many thanks to the staff, especially Rolf Fitzner, at DAAD, for the provided support, flexibility and strong belief in the success of this somewhat novel mission. Thanks also to the staff at the University of Cambridge, the Computer Lab and my college, "Trinity Hall", especially Lise Gough, Fiona Billingsley and Julie Powley, who were very helpful.

Of course, being thrown into the cold water of a new code-base is always demanding, however, being thrown into a research group like the "Netos" crew more than makes up for it! Thanks a lot to: Steven Smith, interesting discussions and patience; Mark Williamson, support with the scheduling api and general Xen-internals; Keir Fraser, Xen-Guru; Steven Hand, for lots of critical, helpful comments and supervision; Ian Pratt, making it all possible, additional funding, supervision, a never-ending assortment of new ideas and "try-this-out"s and strong belief in my skills; all PhD students and research associates from FW01, FN01 and FN07, for making this a great time!

This research internship would not have been possible without the help of Hermann Härtig, who introduced me to the Netos-group in Cambridge, supported my application to DAAD and sparked my interest in advanced system's topics. And of course for accounting this paper as my "Großer Beleg".

Additionally, I'd like to thank every brave soul, who took the time to read this paper and made valuable comments on content, style, grammar and spelling, especially to Jean Wolter.

Finally, a thank you to my family, who have been supporting me ever since and made just about everything possible!

# 1 Introduction

## 1.1 Motivation

Virtualisation on commodity hardware has matured from a mere proof-of-concept over an usable implementation to run a single guest OS for testing, debugging and convenience purposes to a stable solution used in large commercial systems, where each physical machine runs multiple guests, offering virtual dedicated serves, server consolidation and additional security.

As soon as more than one guest system is to be virtualised, the question of "Which guest can use which CPU for how long?" needs to be answered. A simple solution would be to leave this question to the layer below the virtual-machine monitor, usually the host OS, which treats guests as normal tasks in the system. However, with Xen's *hypervisor* approach, there is no such thing as an OS-layer below the VMM, creating demand for an own solution inside Xen.

When computing time is to be sold as a service, as it happens with web-hosters and computing centres, accurate accounting and strict allocation of CPU-time is mandatory, further asking for an own scheduler in Xen, preferably with the flexibility to provide firm guarantees[1]. Services, which do not require firm guarantees, but would rather receive a constant share of available bandwidth[2] should be supported, too.

## 1.2 Goals

The goal of this paper is to introduce a new scheduler for the Xen virtual-machine monitor, with the following characteristics:

**Performance** The scheduler should not impair the systems performance without good cause to do so. Instead, it shall support a variety of workloads and provide them with adequate service.

**Predictability** The scheduler shall give predictable service to the guests running in the system, especially in the presence of guests with complementary characteristics and requests, and with un-

[1] with a premium price tag, of course

[2] thus paying considerably less money

trusted and possibly malicious intentions.

**Ease of use** Achieving these goals should not limit the choice of software which can run on the system or require a deep analysis of characteristics (such as execution time distribution, I/O behaviour) to be done in advance. Parameters controlling the behaviour of the scheduler are to have an obvious impact on how they affect the scheduling.

**N:M scheduling** Finally, the scheduler should provide support for running guests with multiple virtual CPUs on hosts, which themselves are equipped with several execution units (such as SMPs, multi-cores and SMTs).

The paper also shows some of the decisions made during the development, how the design was refined and extended over time and gives intermediate results for various implementation details, to highlight the effects of various ways to tackle a single problem.

## 1.3 Outline

Most of the development in this paper has already been done at the University of Cambridge, at the System's Research Group in the Department for Computer Science, during an visiting research internship between October 2004 and August 2005. Because of involvement into other study-affairs back in TU Dresden, it took quite a while to compile this report. This means, that this report deals with Xen's unstable tree of about mid-August 2005, for example change-set 6302:40d68c7d62d0, from `http://xenbits.xensource.com/xen-unstable.hg`

This paper consists of four chapters and two appendices, where the chapters are written in chronological order, reflecting the progress of development, more precisely:

**Chapter 1** is the chapter you are currently reading. It introduces this paper.

**Chapter 2** gives a brief overview about virtualisation, Xen and scheduling. If you're familiar with those terms, you can possibly skip or skim it.

**Chapter 3** focuses on the development of the single CPU scheduler SEDF, which provides guests with firm contracts and bandwidth guarantees, and introduces Xen's scheduling code and various schedulers used by Xen. It points out, how decisions in the scheduler affect performance of the system and highlights a high-performance solution.

**Chapter 4** introduces various extensions to the single CPU scheduler and introduces a balancer, which

ensures predictable performance for guests in the system in the case of changing CPU utilisation and uneven partitions.

**Appendix A** shows some raw measurement data for experiments run in chapter 3, omitted there for lack of space.

**Appendix B** will show some of the key data-structures used in Xen's scheduling code.

Chapters 3 and 4 each contain their own summary section (3.4 and 4.5) and results (3.3 and 4.3).

# 2 Background

## 2.1 Virtualisation

Virtualisation is a technique that multiplexes important system hardware (such as CPUs, hard-drives, network adaptors), thus allowing the user to run multiple operating systems with associated applications on a single machine, each in its own *virtual machine*. In order to achieve this, operating systems must be deprived of their rights to manipulate the hardware directly, as otherwise isolation between the virtualised systems could not be ensured. Two approaches are quite common to tackle the problem of virtualising the most important system resource, the CPU: emulation and (full) virtualisation.

**Emulation** generates an abstract device model of the virtualised hardware in software and reproduces the effects of operation on the device. For the example of a processor this results in a model of the CPU, which emulates the effects of the machine instructions. The means to apply the original machine-code to the model are *interpretation*, where instructions are emulated one by one on the fly by using appropriate operators on the model, and *compilation*, where the mapping from machine-code to model-specific operators is done in advance.

Latter clearly has benefits, if code is executed over and over again, such as in loops and functions used often, but suffers from long delays because of compilation. A common trade-off is to find "hot-spots" of code, *i.e.*, those parts of the code that will benefit most from optimisation. Finding those "hot-spots" is done during the execution of the programme, yielding so called dynamic compilation or hot-spot VMs.

Major advantage for emulation is the great flexibility for choosing the architecture of the emulated and the host hardware. For example a system based on x86 architecture can emulate a system based on Z80 hardware[1]

It is quite clear that this flexibility comes with a price and this price is performance. Straightforward interpretation effectively replaces each machine instruction of the virtual CPU with a collection of instructions on the host system, quite often with additional overhead introduced by calls to functions,

[1] This is in fact done, see [Web] and various other emulators for gaming consoles, which themselves can also run on different hardware, e.g. Game-Boy emulator on mobile phone

cache misses and additional branches. For the well-known "QEMU" on x86 hardware, performance for the emulated x86 system is somewhere between $1/4$ and $1/20$ of native performance[Bel05]. Dynamic recompilation can effectively reduce this overhead but creates very complex code for the emulator.

As our main concern is not the emulation of arbitrary hardware, but rather a replication of the existing system, the overhead and performance penalty of emulation do not seem to be justified. Especially when building a proper model for a modern x86 CPU with surrounding hardware itself becomes a difficult problem, not mentioning the performance impact, which occurs when running it.

**Virtualisation** The resort from that is quite obvious: Don't emulate a similar CPU but rather run the code directly on the host CPU. This is the major idea of virtualisation and the starting point for this paper. However, handing over control of the CPU to the emulated (also called *guest*) code has serious implications on system security, as measures have to be taken to prevent the guest from overtaking the system, isolate guests among each other and protect the virtualisation software (usually called the *virtual machine monitor* or VMM) from the guest.

Fortunately most modern CPUs are equipped with a multilevel protection architecture, which prevents running code from using and modifying privileged CPU resources such as page-tables, mode flags, ... Attempts to use such resources by the guest code usually create an exception which invokes a handler of the VMM. This handler checks the validity of the original request causing the exception and, when valid, emulates the effect and returns to the guest, always ensuring that it cannot gain excess privileges or interact with the monitor or other guests in the system.

Virtualisation itself is not a new idea, early systems such as IBM's S/370 have been using this approach since the early nineteen-seventies [Cre81] to partition a large server machine for multiple virtual hosts.

Isolation of virtual machines is essential to virtualisation, as the operating systems running inside the virtual machines are not aware of the underlying virtual machine monitor and assume full control over crucial system resources of the processor (such as page-tables, operation mode flags) and are therefore likely to modify these resources.

**X86 hardware** As hardware based on the x86 architecture has become increasingly powerful and thus the *de facto* standard in personal computing and also conquers low- to mid-range servers, various attempts have been made to use virtualisation on this hardware architecture. But virtualisation has not been on the agenda for the early instruction set architects, support for isolation of processes has been integrated into the CPU since the Intel 286 [Int97], with various refinements in subsequent processors.

Several flaws in the x86 architecture (*e.g.* [RI00]), however, cause problems for virtualisation, as they

do not cause an exception when they are executed with insufficient privileges, but rather fail silently and reveal information. Therefore the VMM does not get a chance to check and emulate the instruction.

Therefore attempts on virtualisation of x86 hardware often use a technique called *binary rewriting*, that modifies the binary code of the guest operating system and replaces dangerous instructions with traps into the machine monitor. Care has to be taken as code in the guest might be (intentionally) obfuscated, interleaved with data and check-summed, therefore allowing the detection of the modification or errors during code modification. Various heuristics and techniques are used in the products (such as VMWare[VMw] or VirtualPC[Mic]) commercially available for virtualisation, that are tuned to specific operating systems.

The introduction of additional overhead, because of missing hardware support and the general mechanism of traps on privileged instructions causes a notably performance overhead, for example resulting in only 30% performance for VMWare Workstation 3.2 running SPEC WEB99, when compared to native performance [BDF+03], thereby making large scale virtualisation for commercial purposes (such as virtual hosting) uneconomic.

The cause of the problem is the initial assumption that the guest should not be aware of the fact that it is virtualised, or putting it differently that the virtualisation attempts to produce an exact copy of the underlying hardware.

**Paravirtualisation** Relaxing the initial requirement of providing each guest with an exact copy of the system interface to offering just a similar one is the key idea for paravirtualisation. This of course means that modifications of the guest operating system have to be made, but as the provided interface is similar to the original one, this becomes feasible (about 3000 lines of changed code for Linux 2.4 [BDF+03])

This technique has been called *paravirtualisation* [WSG02], the VMM does not have to perform binary rewriting, as the silently faulting instructions are replaced by traps into the VMM. Additionally knowing about virtualisation inside the guest operating system helps to increase performance, as legacy mechanisms such as interrupts can be replaced with faster and more suitable primitives, such as lightweight event notifications. For various other resources, such as time, memory and caches, it is beneficial to also know about the real situation, as the guest operating system may then support cache-colouring and provide better service to timed resources, such as TCP/IP timeouts.

## 2.2 Scheduling

Once a resource (such as the CPU) is used by multiple entities (usually called *jobs* or *threads*), there has to be some entity, the *scheduler*, which controls access to the resource. Depending on the type of the resource, the surrounding environment and requirements for the resulting schedule, one can find various ways of classification for scheduler operations.

**Interactiveness** For a fixed workload with fixed timing information, a schedule may be computed *offline* before the actual distribution of resources, thus allowing for extensive re-orderings and optimisations. The result will be a simple plan that specifies for each point in time the jobs, which will receive the resource, making decisions trivial during the actual execution. An example for such a schedule would be a manufacturing line in a car factory, where decisions have to be made as when to assemble which part.

Under more unsettled conditions as found in interactive systems, such a computation of a schedule beforehand is not possible. The scheduler has to make its decisions *on-line*, during the actual execution of the jobs, taking current events into account and interactively allocating the resource to the jobs. Decisions are made by obeying a scheduling policy, which gives a more general rule set than the full precomputed schedule used above.

**Revocability of resources / preemptiveness of jobs** Resources and jobs can offer varying support for interruption of access to the resource and execution of the job. Hard-drives and the according I/O-requests cannot be interrupted once the request has been forwarded to the disk, as the disk is non-revocable, making the request non-preemptive.

Given a revocable resource, such as most modern CPUs, the scheduler can preempt the execution of the currently running job and select another runnable job at its discretion. But as preemption may come with an assigned penalty, the scheduler however can also decide to wait for voluntary clearance of the resource by the currently running job.

**Real-time behaviour** Scheduling can give various levels of guarantees to the scheduled jobs, such as fixed completion time, probabilities for finishing times / amount of finished work by a point in time or just a general fairness guarantee. Schedulers which are capable of making guarantees according to timing are usually called *real-time* schedulers, with various sub-classifications. Other schedulers do not make strong guarantees, but rather optimise for other criteria, such as throughput, utilisation, latency or fairness.

**Multiple resources** In order to increase performance, resources can be multiplied. Scheduling now not only has to specify when to execute which job, but also has to determine on which resource. This might become difficult, as some resources imply penalties for jobs that switch to another copy of the resource, for example in a multiprocessor system a CPU can have cold L1/L2 caches, when a thread is moved there from another CPU, thus causing longer delays during memory accesses, as the cache-lines do not contain the right memory locations yet. Other problems might occur, when copies of resources are not independent of each other but for example interfere in terms of performance with each other.

The scheduler ideally knows about the multiple resources and their interrelationships, and tries to avoid penalties as far as possible and maximise benefit from multiplying the resource.

**Scheduling the CPU** This is the main concern of this paper, as it is the system's key resource. As such, all modern operating systems multiplex the CPU among the runnable (usually called *ready*) threads into time-slices according to some policy, usually used to create an illusion of parallel execution of multiple applications for the user on interactive systems, to reduce delays and to interleave threads with different timing requirements.

Usually scheduling consists of a policy and a component inside the operating system's kernel, which enforces this policy, by pausing the execution of the currently running thread and selecting another thread from the set of ready threads. As most modern CPUs are revocable resources, *preemptive* scheduling can occur, usually triggered by an external source, such as an configurable timer. But in recent history some operating systems still used *non-preemptive* scheduling mainly for the sake of reduced complexity and backwards compatibility, thus depending on good-natured and bug-free threads.

However relying solely on threads to give up their place on the CPU voluntarily can cause *starvation*, a scenario, where ready threads will never get any CPU-time, if other threads behave badly. This is the main reason, why all modern general purpose operating systems incorporate a preemptive scheduler that enforces the scheduling policy preemptively. Threads usually can however, give-up their time slice, indicating that they do not have anything useful to do anymore, allowing the scheduler to select another candidate at its discretion even before the usual scheduling event.

**Scheduling for virtualisation** Today's computing power easily allows for execution of multiple guests, each in it's own virtual machine. The virtual machine monitor now has the duty to execute them on the given number of real CPUs, similar to the operating systems scheduler that deals with the threads, just one level below.

## 2.3 Xen

Xen[BDF+03] is an open-source VMM originally developed at the University of Cambridge. It offers low-overhead virtualisation by using the aforementioned paravirtualisation technique, using modified kernels for Linux, *BSD, Plan9, Minix, OpenSolaris and others. Additionally it supports unmodified guest systems on hardware with virtualisation support (namely Intel's VT [Int] and AMD's Pacifica [AMD]) and provides them with a device model.

Guests are running in virtual machines, called *domains*, the actual VMM of the project is called *hypervisor* and traps into it are dubbed *hypercalls*. Hardware access is restricted for normal unprivileged domains, an privileged domain *dom0* exists, which has full access to the hardware and contains all device drivers. Access to the hardware by other domains is realised via an asynchronous notification mechanism and ring buffer of shared memory [BDF+03] an overview of the system's architecture is given in figure 2.1.

Xen runs on x86 hardware with support for PAE, x86_64, virtualisation such as VT and Pacifica. There also exists an IA64 port. Xen can be used on SMP machines and also supports multiple *virtual CPUs* for each domain, enabling the user to run SMP operating systems inside domains. The numbers of physical CPUs and virtual CPUs are independent from each other, theoretically allowing to run a guest OS for 32 CPUs on a single processor machine. The given architecture poses some similarity to classical operating system's architecture with domains being similar to *processes* and VCPUs to *threads*.

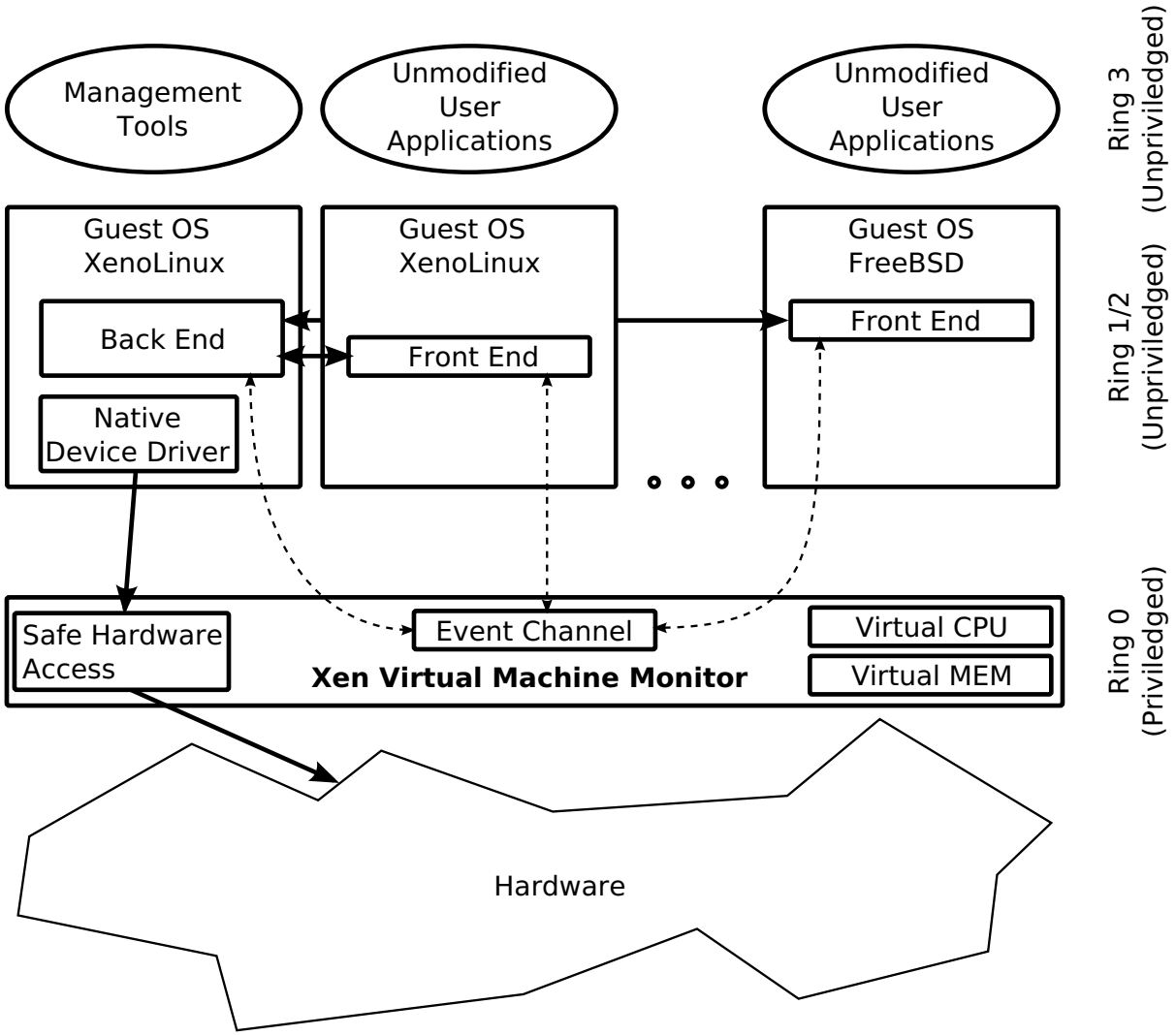


Figure 2.1: Overview of Xen's architecture. Solid arrows between front-end and back-end drivers indicate data-transfer (through ring-buffers), dashed ones mark event notifications for ring-buffer access.

# 3 Scheduling on a single processor

## 3.1 Overview

Scheduling on a single processor is the basic problem for any scheduler and served as a starting point for me to get acquaintance with Xen's architecture in respect to scheduling. Xen's architecture has been and still is changing quite rapidly, incorporating new features and fixing bugs, especially in the *unstable* tree, which I used for development. For example the support for symmetric multiprocessor aware (SMP-aware) guests has been included, changing data-structures used for scheduling tremendously. Various fixes and extensions have been added, so I had to adapt my version of the code accordingly.

### 3.1.1 Types of workloads

Various workloads exist in general purpose operating systems, some have soft-real-time characteristics, requiring a (varying) amount of time periodically, such as media-players decoding multimedia data, releasing the CPU once they have filled their buffers of decoded media.

Others, such as web servers and databases are I/O-bound workloads, which spend a lot of time waiting for I/O requests to disks, network cards and other peripheral devices to complete. They usually release the CPU, when waiting for the requested device. However their performance is heavily dependent on how quickly they get scheduled on the CPU again, once the device has signaled completion of the requested operation. A third kind of workload is the CPU-bound workload, whose main resource is the CPU. A programme calculating Pi to the n-th digit would be such an example. Threads in this class are pretty pliable to CPU-scheduling, as their performance mostly depends on the total amount of time spent on the CPU (within reasonable bounds, neglecting penalties from task-switching overhead, cold caches, *etc.* for the moment).

As the world (unfortunately) is not black and white (and well, not RGB), many real applications do not strictly belong to one of the categories, but rather present a mixed behaviour of two or all three characteristics, presented above. Typical examples can be found everywhere, to name a few: media-players often need I/O to get access to the data to decode, can have post-processing subsystems that "eat

up" available CPU-time to improve image quality. Compile operations contain I/O based parts (access to the source file, writing the binary) and also need plain CPU-time to do their code translation and optimisation steps.

As scheduled entities in Xen are entire operating systems, each including potentially many different user applications, this mixed workload-characteristic is likely to apply aswell, even with stronger fluctuations and peculiarities.

As a result, domains and VCPUs in Xen cannot easily be modelled by the *periodic task model* as described in [LL73] and [Liu00a], moreover we will use a mixture of periodic execution and the according metrics (periods and execution times, called slices in this document) and aperiodic execution (including servers, budgets and the like, depending on what actions the domain is performing.

### 3.1.2 General scheduling code in Xen

Xen features a pluggable scheduler architecture, allowing it to use special schedulers for different scenarios of operation, switchable at boot time for the whole system. This section will give a brief overview of the interface used by all schedulers.

Each domain is represented by the `domain` structure, which contains information for memory management, locks for access, information for event-channels, an array of all related *virtual CPUs*, short *VCPUs* and holds a reference to private information for the scheduler. See Appendix B.1 for a detailed listing.

VCPUs are used by the guest OS just as normal CPUs, *i.e.*, the guest schedules all threads running inside of it to run on the VCPU. For Xen the VCPUs are the units it has to place on the existing physical CPUs. For the beginning let us assume that there is only one physical CPU existent in the system, or equally, that we treat each physical CPU in the system separately, each with its own set of runnable domains and VCPUs, relying on the administrator of the system to assign domains and their VCPUs to the physical processors. The `vcpu` structure (see Appendix B.2) thus contains all information for scheduling, such as the status flags, times each domain has been scheduled and a map, on which physical CPUs this virtual CPU is allowed to run.

The interface to each scheduler is defined in `struct scheduler`, see Appendix B.3, and contains functions to control the domain life cycle (such as `alloc_task`, `add_task` and `free_task`), to change the state of a VCPU / domain (`sleep` and `wake`) and to adjust scheduling properties of a domain (`adjdom` and `notify_pincpu`). The most important function of the scheduler is `do_schedule`, which is called periodically and returns the next domain to run and a time step on when to invoke this method again.

### 3.1.3 Scheduler interface

**Block** VCPUs block, once the guest operating system's scheduler does not find any runnable task inside the guest operating system, that is the case when the guest runs its idle task. The paravirtualised guest uses `HYPER- VISOR_block` to signal that it is waiting for an event to unblock it.

The scheduling hypercall is forwarded to `do_sched_op` in `schedule.c` by Xen's entry code. The called function `do_block` sets the blocked flag for the VCPU and checks for events one last time. If none are pending, the scheduler is invoked to signal the blocking of the VCPU and another runnable VCPU is scheduled by `enter_scheduler`.

See figure 3.1 for a display of the call-graph for the involved functions.

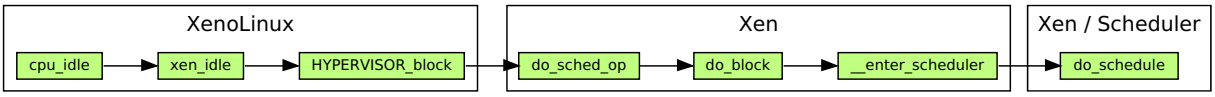


Figure 3.1: Call-graph for the "block" function of the scheduler.

**Unblock** Once a VCPU is blocked, it can be awoken with `vcpu_unblock` (in `sched.h`) which calls `vcpu_wake` and notifies the scheduler via the function `sched_op_wake`. As can be seen in figure 3.2, there are various reasons to wake up a VCPU. Most prominently of course is the arrival of an IRQ.

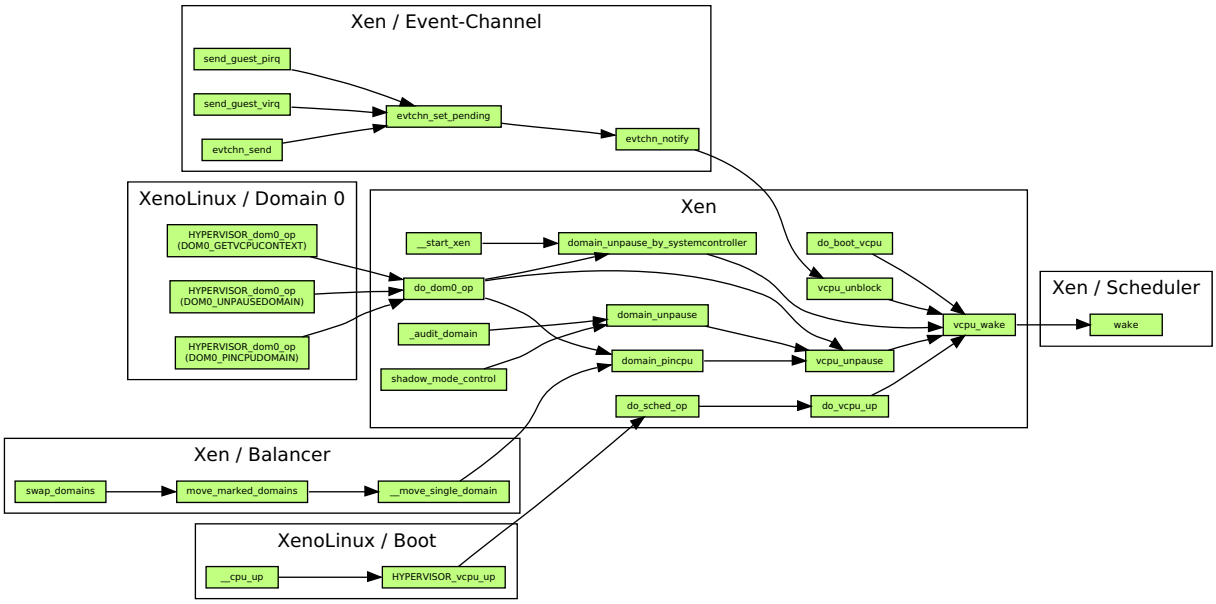


Figure 3.2: Call-graph for the "wake" function of the scheduler, which unblocks blocked domains / VCPUs and wakes up sleeping ones.

All such events are sent through an event channel, an entity that allows to connect various domains and their VCPUs. Together with the asynchronous ring buffers, they are the main [FHN+04] form of communication between domains and VCPUs and allow for fast, low-overhead transfer of data between domains.

Additional mechanisms, such as timers can also be used to wake up VCPUs. They are represented as virtual IRQs (*VIRQs*) and finally represented as events in the event channel.

**Sleep and wake** In order to halt a VCPU, the sleep operation is used. It differs from the aforementioned block primitive in two key ideas: Firstly, it can pause any VCPU present in the system, whereas only the currently running VCPU can block. Secondly a sleeping VCPU is not woken up when there are pending events (such as IRQs or ringbuffer notifications) available for it. In fact sleeping freezes a VCPU entirely.

As can be seen in figure 3.3, various paths use the sleep operation. Common uses are the `domain_pause` and `vcpu_pause` functions, together with their counterparts `domain_unpause` and `vcpu_unpause`, their position in the call-chain is visualised in figure 3.2. Pausing and unpausing a domain/VCPU adds a counter, which keeps track on the number of pause/unpause operations and allows the domain/VCPU to run only when their counts are even.

From the point of the scheduler blocking, sleeping and pausing are usually equal operations. They all change the state of the VCPU such that it cannot be scheduled (observed by `domain_runnable(vcpu v)` returning false) until reactivated, which is done by the `wake` operation.

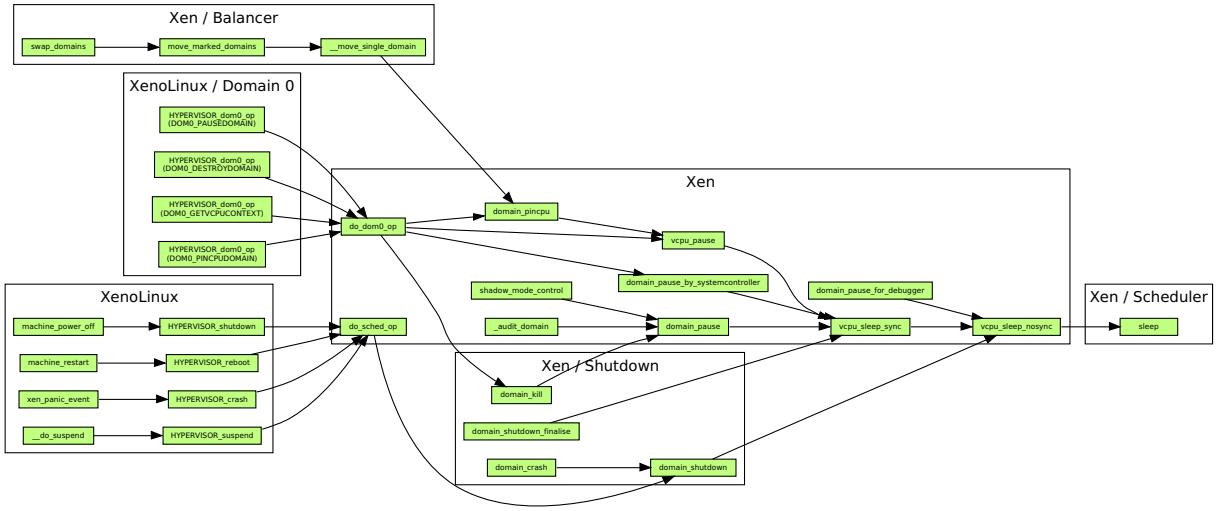


Figure 3.3: Call-graph for the “sleep” operation, which freezes domains / VCPUs.

**Yield** If a guest decides it does not need the CPU anymore for the current allocation, it can release its given allocation and thus make unneeded CPU-time available to the system. Using this function is straightforward, the guest calls `HYPERVISOR_yield`, which causes a trap into the hypervisor that eventually ends up in `schedule.c`, more specifically in `do_yield` and just causes an extra run through the `do_schedule` function of the scheduler, which deschedules the current VCPU and returns another runnable VCPU.

Yielding the CPU is useful for guests that need to poll some resource but do not want to hog the CPU too much. The paravirtualised version of the 2.6 Linux-kernel does not use this this feature widely, but rather uses event-based notification together with blocking, which reschedules the VCPU just when necessary, thus removing the additional overhead of context switches when sitting in a poll-yield loop inside the guest. Other guests, such as *BSD and the 2.4 Linux-kernel and other architectures (IA64) make use of the yield operation more frequently.

See figure 3.4 for the call-graph for the yield-operation.

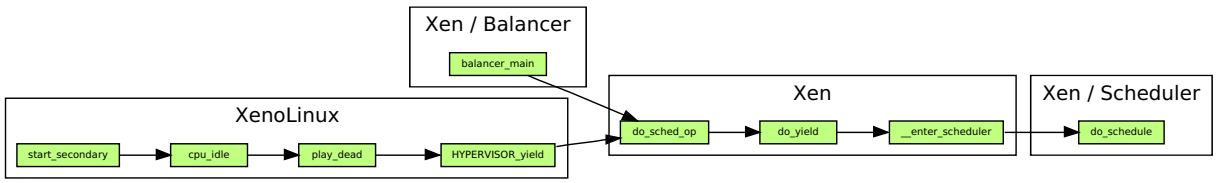


Figure 3.4: Call-graph for the “yield” operation, which releases the current slice of CPU-time. Used rarely in XenoLinux 2.6 on x86, more often in *BSDs and IA64 architecture.

**Scheduling** The scheduler it self is invoked by the generic scheduling function `__enter_scheduler` (in `schedule.c`), which collects statistical values, disarms timers, takes necessary locks and then calls the `do_schedule` operation of the custom scheduler. `do_schedule` identifies the next running VCPU, according to its scheduling policy and also specifies the time of the next upcoming scheduling decision (such as the end of a time-slice, *etc.*).

The generic scheduling function then takes care of rearming the timer with the requested scheduling time-out, further bookkeeping and finally switching contexts to the new VCPU by `context_switch(prev, next)` and `context_ switch_finalise(next)`, both dependant on machine architecture (for x86-architecture they are defined in `arch/x86/domain.c`).

### 3.1.4 Scheduling algorithms in Xen

Xen's pluggable scheduling architecture can incorporate many schedulers. Before I started working on my own scheduler, the following schedulers were present in Xen: *BVT*, the "borrowed-virtual time" scheduler [DC99]; *Atropos*, a soft real-time scheduler and finally *rrobin*, a simple round-robin scheduler.

**BVT** The borrowed-virtual time scheduler is a scheduler which extends the principles of the *SFQ*, "start-time fair queuing", scheduler [GVC97] to VCPUs which are to be scheduled on a CPU. SFQ is mainly concerned with scheduling packets in network switches and routers, where an outgoing port of the switch is the resource to be scheduled among the competing incoming packets.

SFQ is thus handling a non-revocable resource, operating non-preemptively and relies on packets to be of finite length (defined in network protocol specifications). Under these circumstances, it does good job of scheduling according to a fairness metric, by bounding

$$\frac{N_f(t_1,t_2)}{w_f} - \frac{N_g(t_1,t_2)}{w_g} \leq H(f,g)$$

where $f$ and $g$ are network streams, $N_f(t_1,t_2)$ is the total number of transferred bits for stream $f$ in interval $[t_1,t_2]$ and $w_f$ the weight for stream $f$, see [GVC97].

But as already mentioned in subsection 3.1.1, threads/VCPUs in a multi-purpose system have quite different characteristics, especially in terms of priority and latency requirement. The CPU also differs from an outgoing network link, as it usually can be revoked from VCPUs running on it, whereas packets cannot be interleaved on the outgoing network link.

As BVT is dealing with scheduling CPU, it extends the SFQ algorithm by introducing a *context switch allowance*, to avoid trashing the CPU by switching contexts permanently and *warping*, which adds support for soft-real-time tasks, which have to react quickly to incoming I/O.

The general idea of BVT is the notion of *virtual time*, which can simply be seen as consumed CPU-time for each VCPU divided by the VCPU's weight. Scheduling then just selects the VCPU with the lowest virtual time and allows it to consume CPU-time (and thus increase its virtual time), until its virtual time exceeds the minimal virtual time present in the system by the context-switch allowance. The virtual time of unblocking VCPUs is either set to the system's minimum virtual time or kept at its own value, whichever value is greater.

Latter mechanisms ensure that a thread cannot save arbitrarily much virtual time, which it might use at a later point in time to starve the system. In order to give soft-real-time applications a better treatment once they unblock, Duda & Cheriton introduce warping, which offsets the virtual time of a certain VCPU,

thus making it more likely to run instantly when woken up. In order to prevent excessive abuse of this feature, a set of control parameters is introduced (somewhat similar to a time-slice and period setting, in which warping is legal), but the authors admit, that often the default setting, switching this feature off, is suitable.

**Atropos** The Atropos scheduler originates from the *Nemesis* project [LMB+96]. Its main way of specifying the scheduling parameters are the notion of periods and slices. If a VCPU is assigned a period of $p_i$ ms and a slice of $s_i$ ms, the Atropos scheduler will ensure that the VCPU will receive the assigned $s_i$ ms during every interval of $p_i$ ms. In order to determine a schedule, the *earliest deadline-first* (EDF) algorithm [LL73] is used, using the end of the next period for each thread as its deadline. This is identical to the general model of periodic tasks, where deadlines are aligned with period boundaries, too.

The major difference is, that applications / system administrators cannot specify the exact position of the period boundary in the sequence of periods, *i.e.*, the "phase-shift", but their location is rather specified by the time $t_{start}$ of the application, yielding the sequence of $t_{start} + p_i, t_{start} + 2 * p_i, \ldots, t_{start} + n * p_i$. This is not regarded to be much of an issue, as common soft-real-time applications do not require the specification of the absolute position of deadlines[1].

As Atropos uses the EDF algorithm, each set of VCPU with $\sum_{i \in threads} s_i/p_i \leq 1$ can be scheduled with all guarantees holding (in practice a slightly lower bound occurs, due to context switching overheads). It is the responsibility of the system administrator or an external QoS monitoring software component to ensure that this condition is met.

If a VCPU blocks and then unblocks in a new period, its new deadline $t_{dead,i}$ is simply set to $t_{dead,i} := t + p_i$ (with $t$ being the current time) and it receives a new fresh slice. This still retains a feasible schedule, as it just shifts the period to a later point in time and does not increase the load on the CPU. Unfortunately this behaviour can postpone the execution of the VCPU till $t_{dead,i} - s_i$, resulting in sluggish response to interrupts and other events, especially for drivers.

To tackle this problem, an optional *latency hint* parameter $l_i$ can be specified for each VCPU, which modifies the waking-up algorithm: Once the VCPU wakes up in a new period, it's next deadline is set to $t_{dead,i} := t + l_i$, reducing the period temporarily to $l_i$. In order to avoid overloading the CPU, the slice has to be scaled accordingly[2], so the VCPU receives $r_i := s_i * l_i/p_i$ time for the new (shortened) period. This forces the scheduler to run this VCPU within time $l_i$ from the time of unblocking, thus allowing to react in a timely fashion to interrupts and still retains the feasibility of the schedule, at the cost of an

[1] as opposed to embedded systems, such as an engine controller

[2] which is an extension to the original algorithm, described in [LMB+96], where the current reservation $r_i$ would not be scaled and threads/VCPUs could miss their deadlines

increased number of context switches. When the VCPU does not block during this next period, its period and slice are scaled up to reach their original values gradually. The idea of adapting slices and periods will be elaborated further upon, more precisely in subsection 3.2.3.

**Round-robin scheduler** The round robin scheduler is a simple scheduler to demonstrate the use of Xen's scheduling API, and thus does not contain much sophistication. It simply selects the first thread from the queue of runnable thread and lets it run for certain amount of time $t_{slice}$ which can be set by the system's administrator. Once the thread has finished its slice or gives it up voluntarily (by yielding), it will be enqueued at the end of the runnable queue. If the thread blocks, it is removed from the queue, and added to the tail once it unblocks. I/O performance may be increased by putting it to the head of the runnable queue, giving it control over the CPU immediately. However this can lead to starvation for other VCPUs at positions further to the back in the queue.

## 3.2 Creating a new scheduler for Xen

### 3.2.1 Starting point

Given that Xen already contained some schedulers, why would anybody want to build a new one? Xen's manual recommended BVT as scheduler to be used. In fact, BVT was used as the default scheduler in Xen 2.0. As the round-robin scheduler has not been designed with performance in mind, its use in a production system was out of question, leaving the choice between Atropos and BVT. However, Atropos behaved unexpectedly and constantly resisted all attempts to debug it, although the general mode of operation was evident.

Summing it up, BVT was the only functional scheduler in Xen 2.0, but had quite unintuitive ways of specifying weights of domains and contained a lot of parameters for performance tweaking, where the actual results on the resulting schedule and system performance were difficult to predict.

Given this initial situation, we decided to start from scratch and build a new scheduler. Initially, a very simple implementation was planned to make myself familiar with Xen's scheduling API. Given that there already existed a round-robin scheduler, a simple EDF-based scheduler that would give out guarantees to the running VCPUs came to mind, using the general idea of Atropos as an archetype for a very simple scheduler. In order to aid understanding of the process of scheduling and avoid the introduction of bugs, all "fanciness", such as tuned unblocking behaviour was omitted.

Essentially, all VCPUs are set to be able to consume CPU-time everytime, but for distributing available

CPU-time in the system, we divide the (originally continuous) execution of a VCPU $v_i$ into chunks of size $s_i$ and release a new chunk every $p_i$. Old chunks not fully used up are discarded, when a new one is released to the system.

This maps the original execution of $v_i$ to the standard periodic task model.

### 3.2.2 Initial implementation

The initial implementation of the *simple earliest-deadline-first*, SEDF, scheduler manages two queues, namely the runnable queue for VCPUs which still have a valid reservation for their current period and the wait-queue, which contains all VCPUs with satisfied reservations, waiting for the start of the next period.

As the scheduling algorithm is EDF, the runnable queue is sorted by ascending (absolute) deadlines $t_{dead,i}$ of each VCPU, the head of the queue is running on the CPU. If the queue is empty, *i.e.*, all commitments have been fulfilled, the idle-VCPU is selected to run on the CPU. Once a VCPU has consumed all of its time-slice $s_i$ for its current period $p_i$, it is moved to the wait-queue, receives a fresh budget of CPU-time (by reseting the amount of received CPU-time $e_i$ to zero) and a new deadline: $t'_{dead,i} = t_{dead,i} + p_i$.

The wait-queue is sorted in ascending order of the start of each VCPUs next period $t_{sop}$, defined by $t_{sop,i} = t_{dead,i} - p_i$, where $t_{dead,i}$ is the new absolute deadline and $p_i$ the guaranteed time-slice per period for VCPU $i$.

The scheduler operates by selecting the head of the runnable queue to run on the CPU in `sedf_do_schedule`, requesting to be invoked again, once the time-slice of the selected VCPU expires or when the head of the wait-queue starts its new period, whichever happens earlier.

More formally, the scheduler requests to be woken up at time $t_{sched}$ with $t_{sched} = min(t+r_{head(rq)}, t_{sop,head(wq)})$, $r_{head(rq)}$ the remaining portion of current period's guarantee for the VCPU at the head of the runnable queue and $t_{sop,head(wq)}$ the start of the next period for the head of the wait-queue.

As Xen expects the timeout of the scheduler to be relative to now, the scheduler then returns the tuple $(head(rq), t_{sched}-t)$ to Xen, given that the runnable queue is not empty. In case it is, $(id_{idle}, t_{sop,head(wq)}-t)$ is returned. If even the wait-queue does not contain any elements (this happens for idle CPUs and when all VCPUs are blocked), an arbitrary wake-up timer is requested, such as $(id_{idle}, 1s)$.

When `sedf_do_schedule` is called by Xen, it updates the CPU-time consumed by the running VCPU and checks, whether it still has time left from its reservation. If this is not the case, the VCPU is moved to

the wait-queue. Additionally, checks are introduced to deal with VCPUs that are not runnable anymore, *i.e.*, blocking, being halted or destroyed. If `sedf_do_schedule` detects that the current VCPU is not runnable anymore, it is removed from the runnable queue and *not* added to the wait-queue, effectively dropping it from all queues known to the scheduler.

Further operation of `sedf_do_schedule` follows the algorithm described above.

**Blocking, pausing, unblocking and unpausing** As described in 3.1.3, blocking of the current domain consists of setting the VCPU's flags to "blocked" and then invoking `sedf_do_schedule`, which operates as described above. Pausing a running VCPU works essentially the same and pausing a VCPU, which is not running simply removes it from either runnable queue or wait-queue.

Waking up from either blocked or paused state is handled in `sedf_wake`. As this initial version was meant to be as simple as possible, not much interesting things are happening here (as compared to Atropos, which however is buggy). Exact operation is as follows: When a VCPU wakes up, it will be run in the period coming up next with a full slice. More formally, the remaining reserve $r_i$ is set to the full time-slice $s_i$ and the new deadline $d'_i$ is computed to be at the end of the next full period: $d'_i := t_{dead,i} + trunc((t - t_{dead,i})/p_i + 1) * p_i$, where $trunc$ is a function that truncates a floating point number after the decimal point. Essentially, this approach (dubbed *very conservative unblocking*) just continues the sequences of periods during the time the VCPU was blocked and starts the VCPU at the next full period, see figure 3.5 for a graphical representation.

It is obvious, that moving VCPUs to their next full period does not create any overload on the CPU. It is however equally apparent, that the performance could be improved, as VCPUs reacting to an event can be delayed as much as $p_i$, before they actually can process the event. Of course, $p_i$ can be reduced, but not arbitrarily, as small periods entail a lot of context switches even in times where a VCPU is exhibiting CPU-bound behaviour.

**Other functionality** Xen's scheduling API defines various other functions, most of them for the creation / deletion of VCPUs and for scheduler initialisation. It also defines a function `scheduler.adjdom`, which allows changes to the scheduling parameters of all VCPUs running in a domain. With this initial version of the SEDF scheduler, it is possible to set the period and slice of all VCPUs of each CPU to a given value. Changes to these values are made instantly and resulting inconsistencies are fixed up later in `sedf_do_schedule`, once they cause any problems.

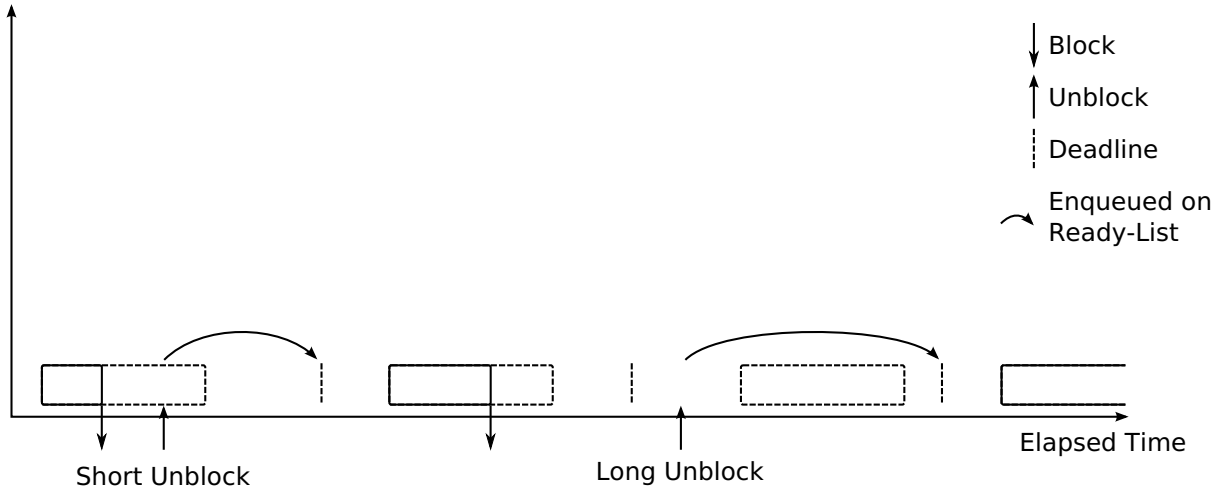


Figure 3.5: Isochronous unblocking scheme with examples for long and short unblocks. Note how far the execution of the VCPU is delayed after the long-unblock, resulting in poor I/O performance.

**Results** Setup for our experiment is the following (and will be used in later experiments for comparison purposes):

- Four domains, each with a single VCPU.
- The three domUs run "slurp"[3], recording and consuming available CPU-time.
- Dom0 is used as an endpoint in a "ttcp"[ttc] benchmark against another machine in the same network.
- Machine is a Xeon (P4 based) server, all domains run on the same physical CPU.
- Network between ttcp partners is a Gigabit-ethernet.

Given the initial focus on simplicity and not performance, one cannot expect the performance to be competitive, especially when VCPUs are I/O-bound in nature. Results for I/O-bound workloads conform to these expectations, CPU-bound workloads however show a nice flat graph for CPU-usage, showing that all guarantees are met exactly, see figure 3.6. Note the exceptionally low throughput of about 38.7 Mbit/s on a (uncongested) Gigabit-Ethernet link, caused by the high latency of dom0s replies to events generated by the card.

Additionally note that guarantees are not exceeded and the CPU is not utilised to its maximum, as the scheduler by now does not have any notion of *work-conservancy*, which refers to the fact, that the scheduler does not idle the CPU as long as there are still runnable VCPUs in the system. By now, there is no means for a VCPU to specify that it can exploit additional CPU-time or even rely solely on available

[3] A tool, which eats up CPU-time and prints out its received share of CPU-time in percent

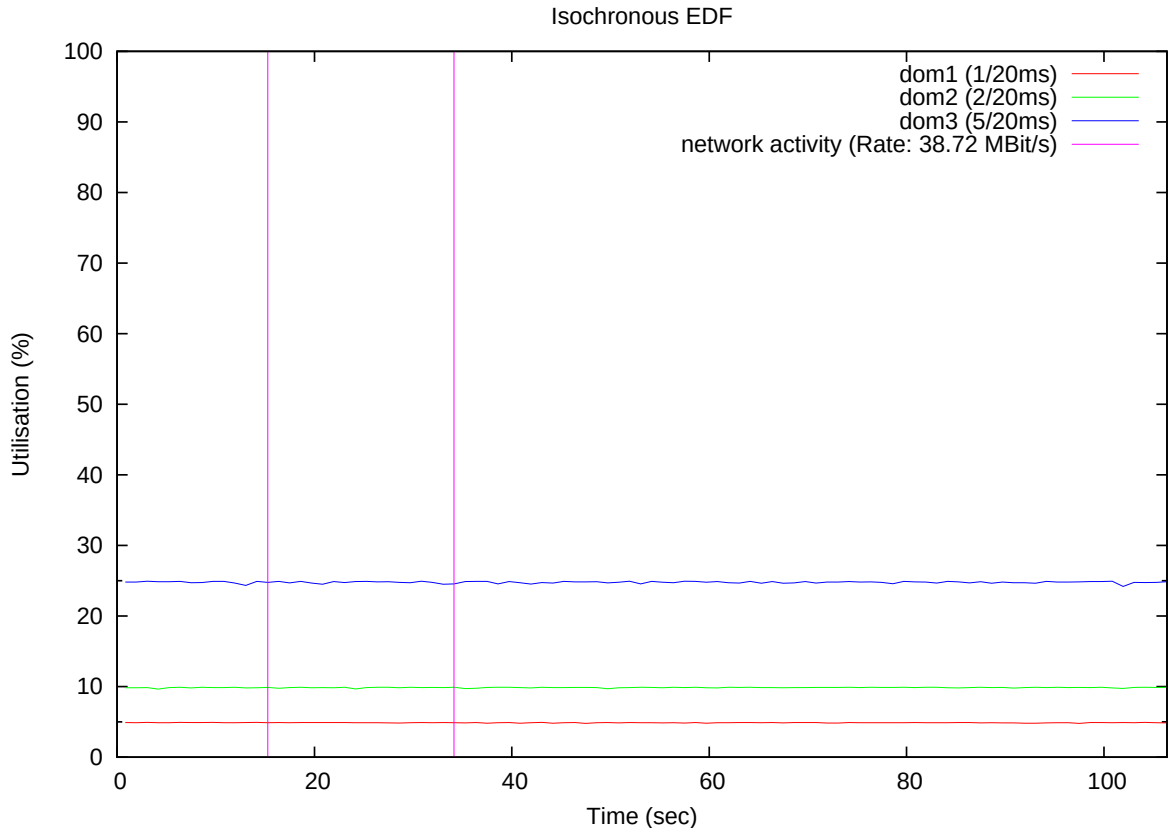


Figure 3.6: CPU-time received by three domains, concurrent I/O marked by vertical bars. Strict isochronous unblocking.

leftover CPU-time. The further development as described in the next two subsections however address both (performance and work-conservancy) issues.

**BVT performance** As a comparison and a goal to reach, I performed the above test with the BVT scheduler (see subsection ), results can be seen in figure 3.7.

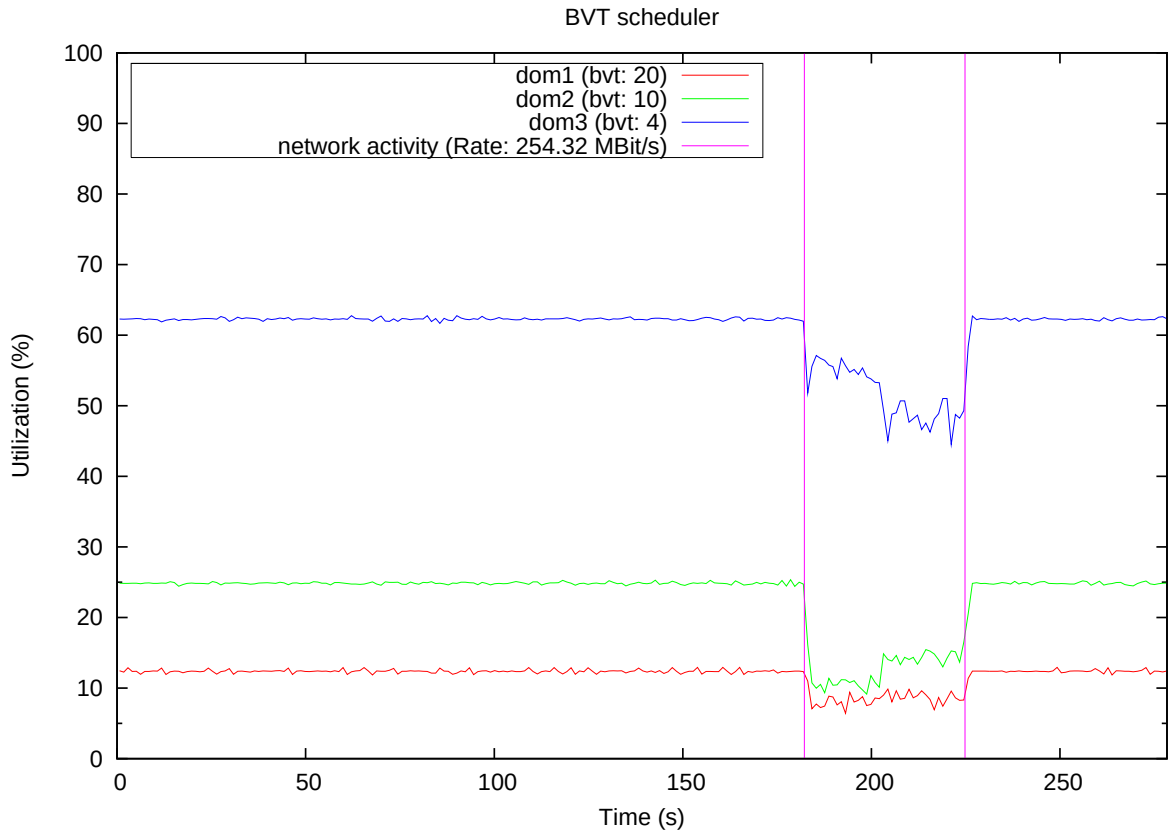


Figure 3.7: Performance of BVT with standard testing scenario. BVT is by default work-conserving.

Settings for BVT have been set so that they result in similar behaviour as in the sedf tests, namely the inverse weight of each domain $w_{inv}$ required for bvt's scheduling algorithm has been set proportionally to each domain's inverse sedf-weight: $w_{inv,i} = p_i/s_i$. Network performance is clearly higher, all idle-time in the system is given to the domains and there are no hard guarantees.

In the following sections, I will introduce tweaks to the SEDF scheduler, improving its performance in comparison to BVT.

### 3.2.3 Sophisticated unblocking

As seen in the previous subsection, a simple deadline-driven scheduler can provide CPU-bound VCPUs with acceptable guarantees. But for Xen's typical fields of application such as server consolidation [Xen] for web servers, databases and firewalls, a fair amount of I/O-activity can be expected from most

domains. Especially for dom0, which contains all hardware device drivers and exports generic interfaces to the other domains present in the system or for the designated driver-domains that just contain drivers for a single hardware device [FHN+04], I/O-performance is crucial.

It is apparent, why the simple algorithm described above does not provide I/O-bound domains with appropriate performance: As soon as a VCPU releases the CPU, because it is waiting for a hardware device to become ready, the earliest point in time it can deal with any response from the device, is the start of the next period. See figure 3.5 for a graphical representation of what might happen.

Given this issue, we have to take a closer look at the guarantees, we want to provide each VCPU running in the system with, as it is impossible to provide a VCPU with its guaranteed CPU-time in a period *e.g.* during which it is blocked entirely, *i.e.*, can not consume it in any reasonable manner. Additionally, trying to fulfill guarantees for I/O-driven domains can make it impossible for other domains to receive their guaranteed share of CPU-time, see figure 3.8 for an example of a set of two VCPUs, with a total utilisation of one (VCPU 1: 10/20 VCPU 2: 3/6), thus schedulable with EDF.

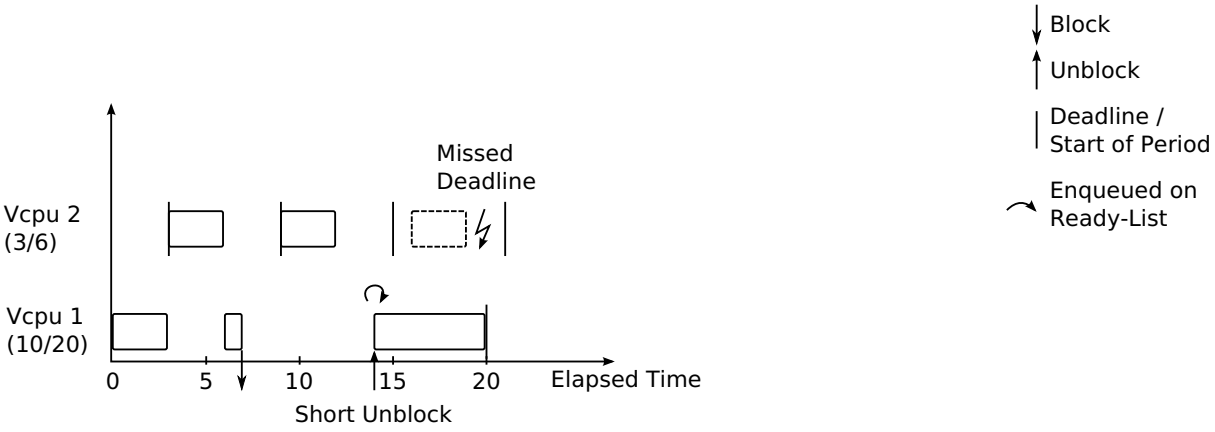


Figure 3.8: Immediately continuing execution of short-unblocking VCPUs can lead to missed guarantees for other VCPUs with strict EDF policy.

This means that in the case of I/O and the resulting blocking (called *self-suspension* by Liu in [Liu00b]), we do not enforce guarantees for these VCPUs. See [Liu00b] for a way to keep guarantees by careful analysis and bounding the amount of time a VCPU can be blocked.

In our scenario, this bound is difficult (if not impossible) to establish and keeping guarantees is not critical. Thus, the general guarantee is changed, such that it is applicable only to VCPUs, that are able to utilise the given CPU-time at any point in time, *i.e.*, that are always runnable. After this restriction, we can try to give I/O-bound VCPUs a service which is as good as possible.

Once we relax the guarantees towards I/O-driven domains, what is it we need to focus on? Let us have a look at a single simple driver that reads data from a mass storage device. Usually, the driver sends a read request to the drive and then waits for a reply containing the data. As mass storage devices usually take several milliseconds to localise the data on the medium, the driver can safely yield the CPU during that time to another runnable VCPU in the system. When the device has localised the data (and read it into internal buffers), it notifies the driver by some event mechanism, usually an IRQ. The driver then processes the data and then probably sends an additional read request to the device for subsequent blocks of data.

Given that the amount of CPU-time, the driver actually needs to create the request and to process the read data, is small in comparison to the devices access delays, its performance is bound by the access time of the device and the delay until the driver is scheduled on the CPU once an event from the device is received. This delay between the incoming request and the actual execution of the drive will be called *latency*.

As the scheduler can not modify the access time of the device[4], it should provide the driver with a low latency.

Please note that this treatment is somewhat opposite to approaches in standard real-time-systems with periodic task models. In periodic task models, the latency is limited by setting the relative deadline of each job accordingly and the time needed for computation by the driver is carefully estimated and a *worst case execution time* is established, guaranteed to the driver each period (which is defined by the length of the relative deadline). This setup of the scheduling conditions guarantees that each latency requirement is met, given the CPU is not overloaded.

The SEDF scheduler can of course be used in the same ways, by simply setting the driver-VCPU's period to the latency requirement and its slice to the worst-case execution time. With the initial unblocking scheme however, this might still lead to problems, as jobs are set runnable just at the beginning of a new period , *i.e.*, a strictly periodic task model is assumed.

In periodic real-time systems, it can be shown(for example in [Liu00c] and [Liu00d]) that all guarantees hold, even if tasks are not strictly periodic, but also if the period (which also is the relative deadline) is not a strict period, but rather a lower bound on the inter-release time for two consecutive jobs in the task, *i.e.*, the time between the two jobs becoming ready [Liu00e].

Translating this to our driver example, the inter-release time is bound (from below) by the minimal access time of the device. If we set the period to this value, we can refill the slice of the driver each time the

[4] various other components of the OS might tackle this problem, by means of caching, grouping of blocks, read-ahead and bursts

event arrives, without worrying about isochronous continuation of the (old) sequence of periods.

See figure 3.9 for a simple visualisation of what happens.

How does this help with the initial issue of unblocking? Well, if we assume, that VCPUs wake up at least a period later than the beginning of their last period, we can simply set their new start of the period to the point of time when they wake up and enqueue them on the ready-queue and thus guaranteeing that their latency is bound (from above) by the period. I will refer to unblocking in a later period than the one where the VCPU blocked, as *long unblocking*, see figure 3.9.

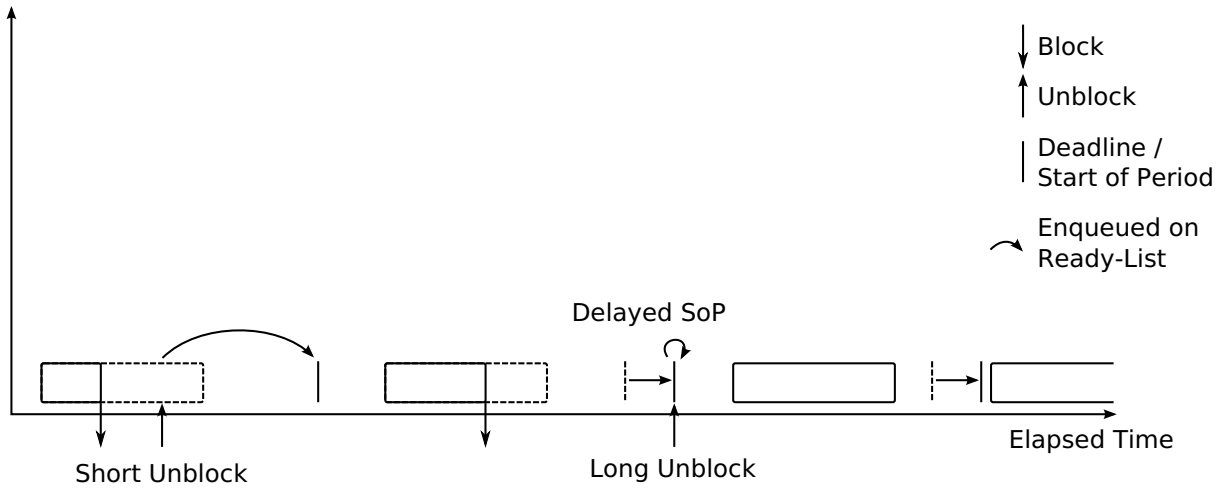


Figure 3.9: Delaying periods allows long-unblocking VCPUs to run straight after unblocking.

We see that dealing with long-unblocking VCPUs is rather straightforward. What happens, when a VCPU unblocks in the same period it blocked in? In strictly periodic real-time systems, this won't happen, by definition of the period being the minimal inter-release time for the VCPU, ensuring that no event can occur before the end of the period, the VCPU blocked in. In Xen however, domains are not just plain drivers and do not access just one device. Setting the period of a domain to low values to meet the inter-release time's condition can mean to sacrifice performance, when the domain is doing more CPU-bound work, as frequent context-switches are introduced by the short periods and slices.

Thus *short unblocking* (figure 3.9) can occur, meaning that a VCPU unblocks before the end of the period in which it blocked. Allowing to put a VCPU on the ready-queue straight away after short-unblocking can lead to overload of the CPU and to missed guarantees by other VCPUs. For now we will deal with this problem by letting the VCPU run at the earliest legal point in time, which is the start of the next period.

This scheme of unblocking, called *conservative unblocking*, is still relatively easy to implement and has better timing characteristics than the originally implemented (isochronous) scheme.

See figure 3.10 for results of a benchmark similar to the one used in figure 3.6. Each domain does not receive more CPU-time than guaranteed. The I/O-operation in dom0 does not affect the other domains, their received CPU-time stays constant. Despite the efforts for supporting I/O, network performance is still unsatisfactory, with a measured rate of just above 39 MBit/s, again on a Gigabit-Ethernet link. This can be explained by the high amount of short-unblocking that occurs (ttcp reports 625 I/O calls per second, resulting in a delay between two calls of 1.64 ms, well below the period-length of 20 ms), which does not benefit from the improved long-unblocking algorithm.

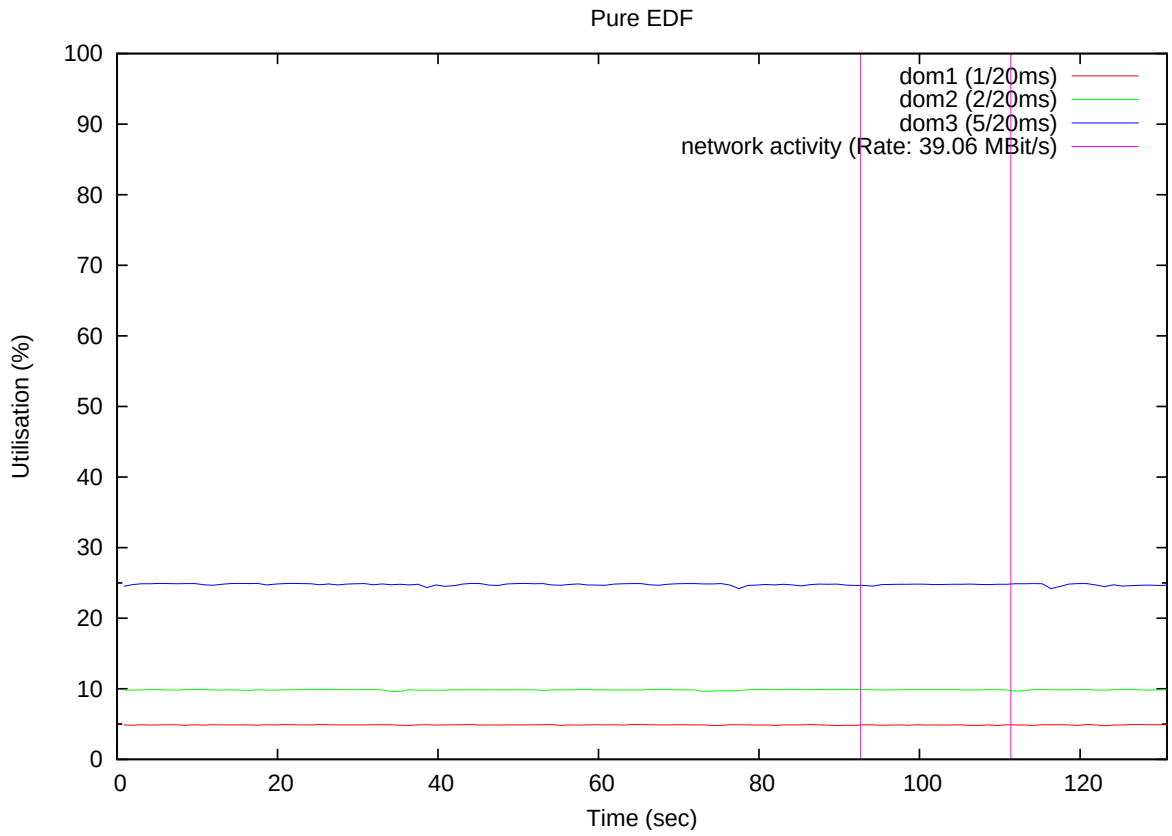


Figure 3.10: CPU-time received by three domains, concurrent I/O marked by vertical bars. Improved long-unblocking.

**Short unblocking** As already mentioned, using a little bit more relaxed scheme for long-unblocking helps, but usually we would want to reduce latency to values below the period, as the period might be set to higher values to reduce the effect of context-switch overhead. Which in turn means that we need to allow earlier readiness for short-unblocking VCPUs.

Although I have stated earlier, that short-unblocking does not allow for earlier execution than the start of the next period, there is a case, where this is possible *without* overloading the CPU. If we treat the time the VCPU was blocked as actually consumed by that VCPU (and to the VCPU that used the CPU during

this time), we can give the remainder of the slice (if there is still one) to the VCPU immediately after unblocking, without causing guarantees to be missed. A simple case is shown in figure 3.11. Again, this unblocking scheme, which I will call *short-resume unblocking*, is simple to implement and does improve the performance substantially.

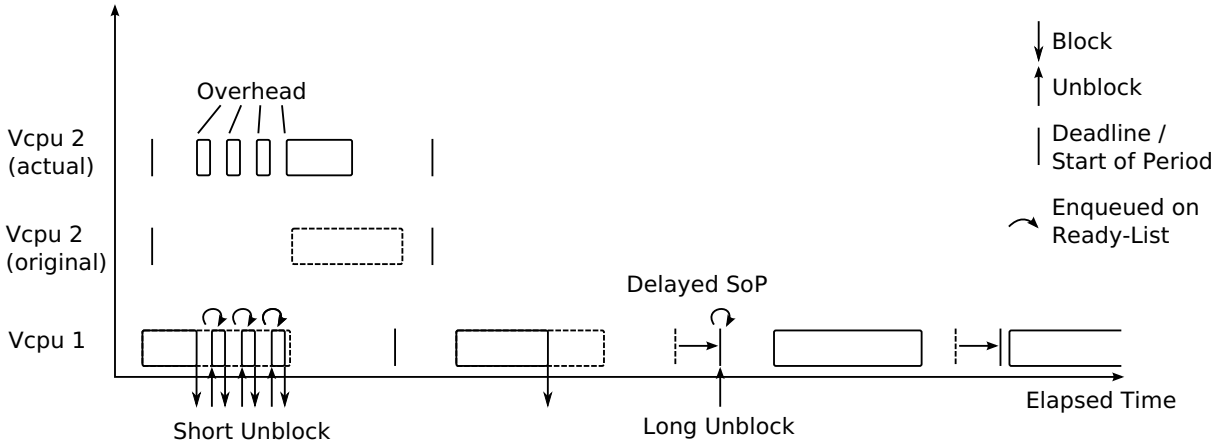


Figure 3.11: Treating blocked time as consumed by a VCPU (VCPU 1 in this example) boosts performance for heavy I/O, but also imposes an increased scheduling overhead for VCPU 2, because its slice can be broken into arbitrarily small pieces.

See figure 3.12. Network performance has increased almost tenfold, to 265 MBit/s. But notice, how the ongoing I/O affects dom3's received CPU-time. This is caused by the increased scheduling overhead due to the many small slices caused by the arriving packets, which results in an increased number of context switches to and from dom0 (with scheduling settings 10ms/20ms), processing the I/O. Dom3 suffers most in this example, because all periods are of the same length and thus VCPUs are scheduled in the same order during the whole experiment. Obviously, dom3's VCPU normally would run right after dom0's VCPU, but is executed interleaved with dom0, because of the blocking and thus has to pay for the scheduling overhead. See sketch 3.11 for an overview of how this looks like in principle.

**Dynamic period adaption** Given the prospect of domains with varying characteristics over time, why do we insist on fixed period and slice settings? Thinking along these lines, one finds the Atropos-scheduler. As already described in subsection 3.2.2, Atropos [LMB+96] originally used a latency hint $l$, which served as a temporary relative deadline, used to boost the priority of an unblocking VCPU reacting to an incoming event, thereby ensuring the driver to actually receive CPU-time at most $l$ after the unblock operation. The authors of the paper admitted that this could lead to missed deadlines for other VCPUs present in the system, however they accepted the degradation, in order to maximise I/O-performance.

But when I looked through the Atropos source present in Xen, another interesting approach had been

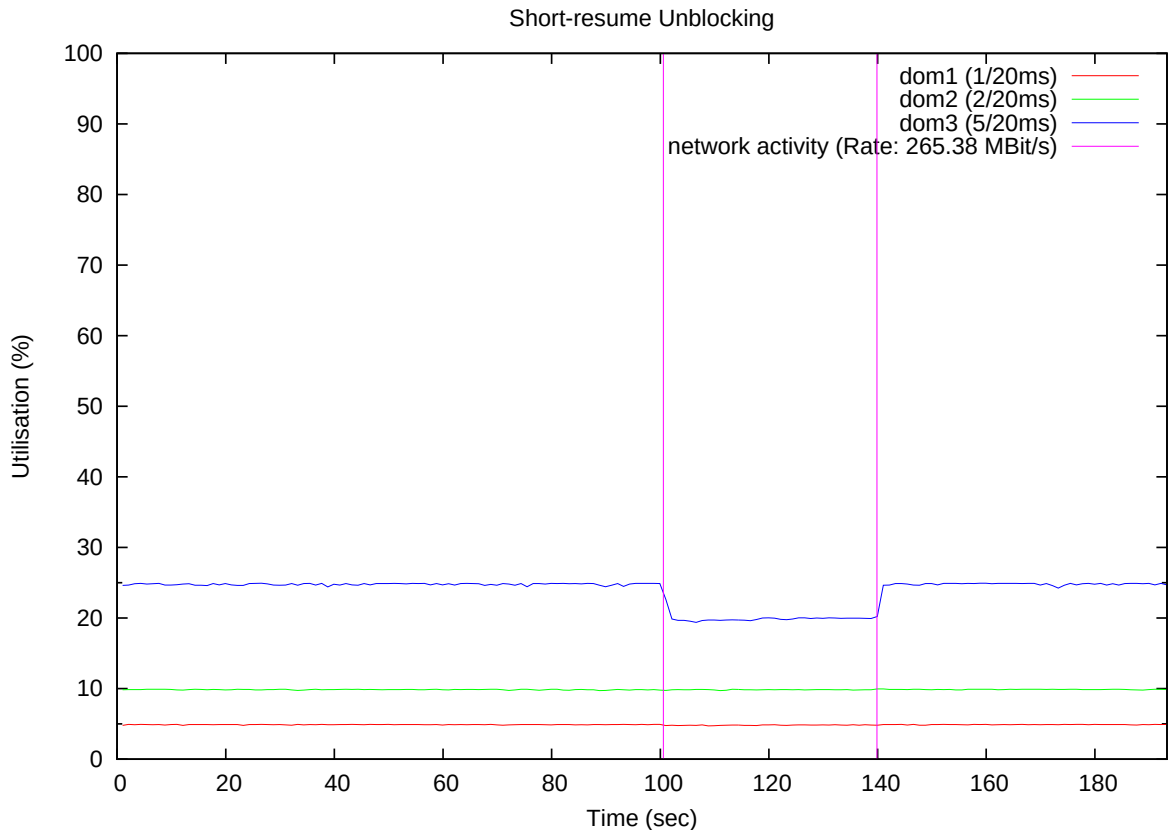


Figure 3.12: Improvement of network throughput with "short-resume" unblocking, at the expense of degraded service for dom3.

already been implemented: Once the period is reduced temporarily, the guaranteed slice for that period is scaled accordingly, thus keeping the overall CPU utilisation constant and still allowing the user to specify a lower latency for domains reacting to incoming I/O-requests. After the period is over and no new blocking/unblocking has occurred, slice and period are increased again with an exponential back-off, simply doubling them until they reach their original values. See figure 3.13 for a simple example on how a VCPU is scheduled according to the latency hint.

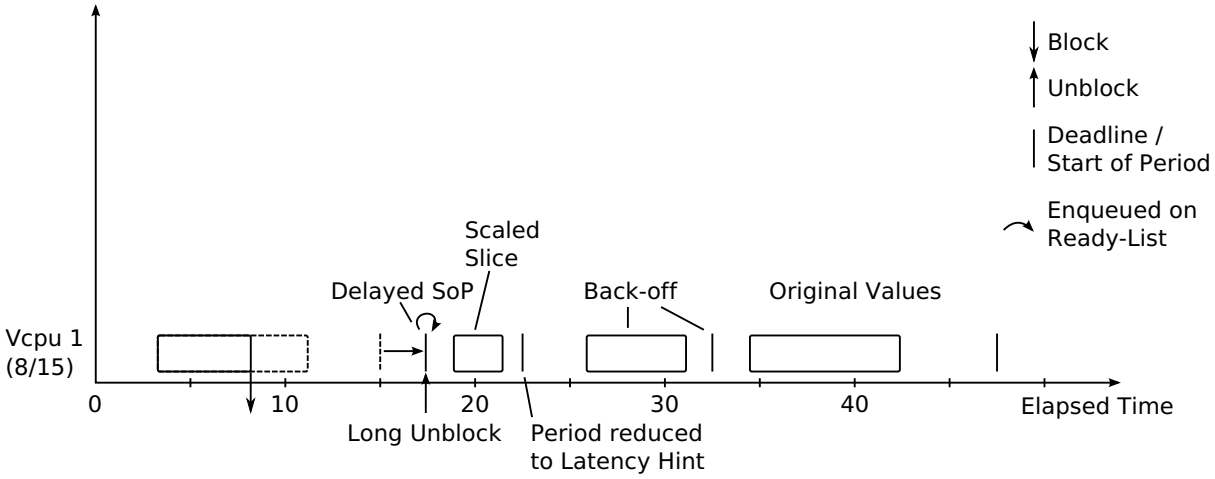


Figure 3.13: Reducing period length after a long-unblock bounds the latency for execution of the VCPU. In the example, the period is reduced to a latency hint of 5 and the slice is scaled accordingly to 8/3. In later periods these values are doubled until they reach the original values.

Decreasing the period and slice of a VCPU can of course only occur at the next period, otherwise there is still a potential overload of the CPU. This means that only long-unblocking is aided with this scheme. But when the period and slice of the VCPU are shortened, more I/O-requests create a long-unblocking condition, resulting in bursts, where the period actually stays at the minimum, which is defined by the latency, or adapts to the device, if that has a lower demands.

A quick implementation in the SEDF scheduler's framework of the same idea did not result in any improvements, of course depending on the amount of latency granted to the I/O-bound domain. But as granted latency was decreased, the effect of frequent context switches was penalising other VCPUs and their guarantees, similar to the results in 3.2.3, where short-resume unblocking would boost I/O-performance at the cost of missed guarantees.

It is to note, that the original approach of the Atropos authors, which aids short-unblocking VCPUs by letting them consume the rest of their time slice was not included in SEDF, as I wanted to provide firm guarantees, which would not have been possible.

**Direct adaption to I/O** An additional issue with Atropos is the needed specification of the latency hint, which might not be easily estimated. Given that, I extended the the idea of adapting periods and slices further towards directly synchronising to the device. Of course, major effects on other VCPUs are to be expected, as the I/O devices can essentially control how much scheduling overhead is in the system.

More precisely, the period is adapted to some value based on the recent I/O-behavior, namely the delay between block and unblock of the VCPU, for example a weighted sum over the last $N$ delays, scaling the slice accordingly. For my quick experiment I just used the last block-unblock delay directly. See figure 3.14 for a graphical explanation of what happens.

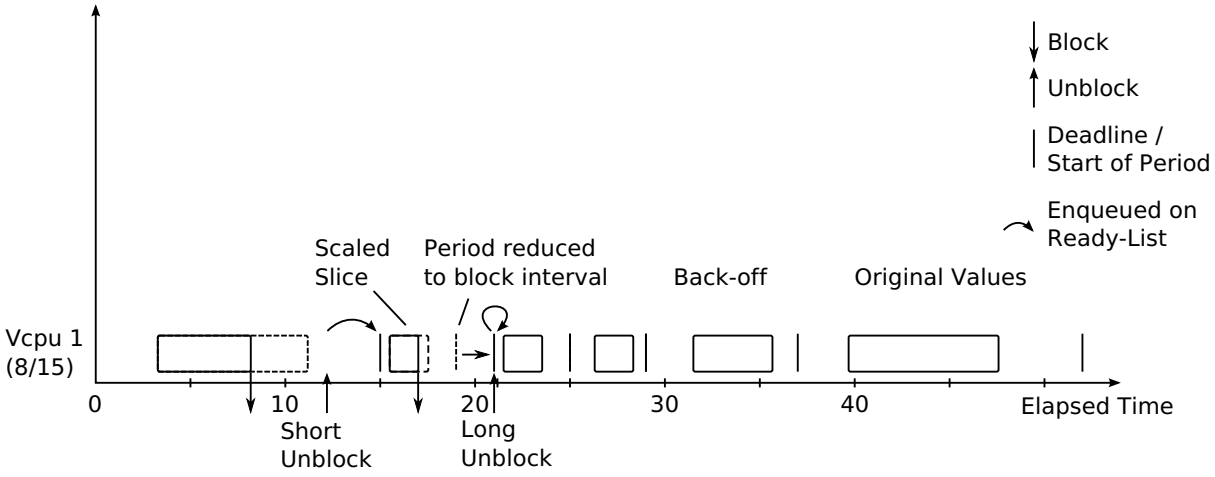


Figure 3.14: Synchronising period-lengths with the rate of I/O will result in instant execution of the handling VCPU. Once the VCPU is in sync with the device, all unblocks will be long-unblocks and thus can be handled immediately. This however allows I/O devices to increase the number of context switches in the system arbitrarily.

Let $t_{block}$ and $t_{unblock}$ be the (absolute) times when the VCPU blocked and unblocked last time, then

$$s'_v = s_v \cdot \frac{t_{unblock} - t_{block}}{p_v} \tag{3.1}$$

$$p'_v = t_{unblock} - t_{block} \tag{3.2}$$

define the new slice and period.

Results (see figure 3.15) were as expected, resulting in about 420 MBit/s with the standard testing scenario, which is the maximum value observed so far. This form of *burst unblocking* however lets all other domains suffer from increased scheduling overhead, as can be seen clearly from the graph.

Given the problems with all the above approaches, how can we ensure high I/O performance, but still provide guarantees to the domains? Obviously, if we could cut down context switching cost and scheduling overhead to zero, the problem would vanish. However, as long as there remains a cost, be it ever so

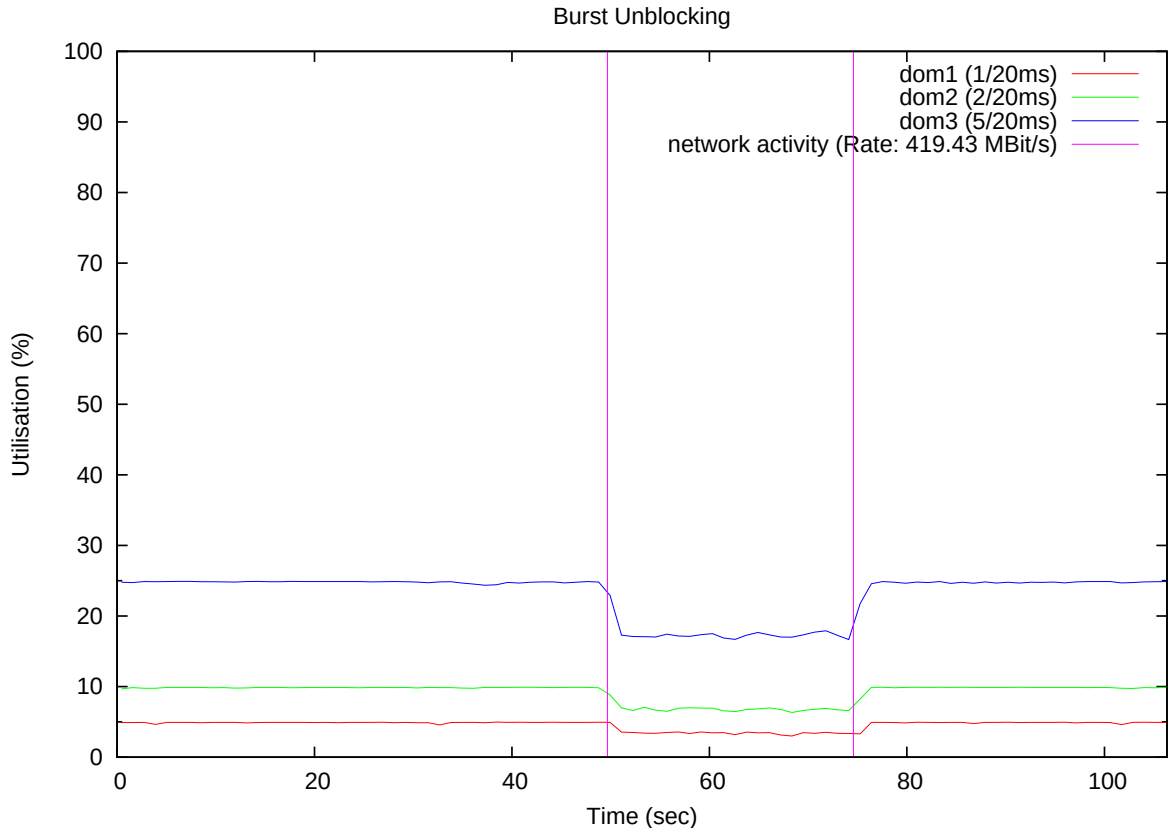


Figure 3.15: Improved network performance, by synchronising periods to rate of I/O. Missed guarantees for all domUs.

small, we just have to increase the rate of I/O and accumulated penalties would hit the system again. Of course, we could incorporate booking tricks, by not accounting overhead to any of the VCPUs present in the system or to the VCPU that caused the I/O, however, this might be difficult to pin down exactly.

The next subsection will have a closer look at spare CPU-time in the system and presents ways of using this available time to support I/O-bound VCPUs.

### 3.2.4 Dealing with slack time

*Slack time* is the time in the system, when all requirements of the runnable VCPUs are fulfilled. See figure 3.16 for an example schedule with marked slack time. Two causes for slack time exist in a system: Underutilisation of the physical CPU by the all the guarantees given to VCPUs , *i.e.*, $\sum_{i \in VCPUs} s_i/p_i < 1$ and occasions where VCPUs do not need their full slice and deliberately yielding it to the system or block on a resource, waiting for a notification from that resource.

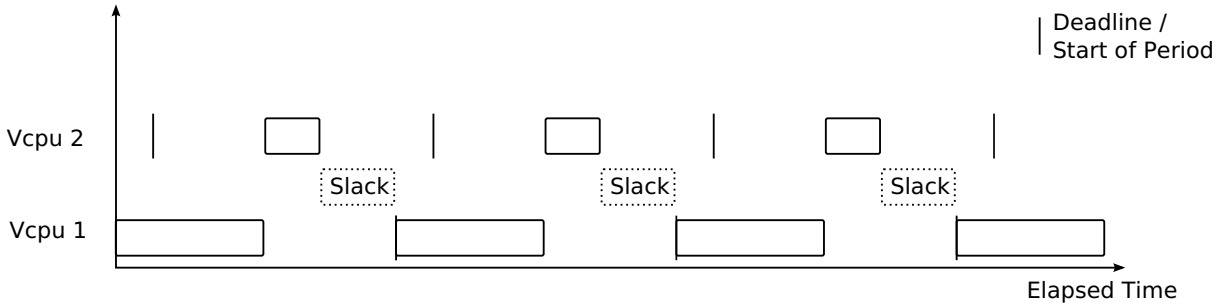


Figure 3.16: Example schedule, with marked slack time available in the system.

**Round-robin distribution** As soon as there is slack time available, there is the question who should get it? And how much? As an initial step, slack time is spread uniformly among those runnable VCPUs flagged by the system administrator to be eligible to receive extra CPU-time, called *extra-aware* VCPUs. All runnable VCPUs that are marked in such a way are stored in an additional queue, called the *extra-queue*. Once there are no more ready VCPUs in the system, *i.e.*, all slices have been used up, the first VCPU from that extra-queue receives a quantum of the available slack time of size $e$, in my experiments of $500\mu s$. In case there is less slack time available, that is, if $sop - t_{now} < e$, slack time will not be distributed, but rather be given to the idle-VCPU. Once a VCPU has used up its extra-time quantum, it is put to the end of the extra-queue, thus distributing extra-time in a round-robin fashion.

Using the testing scenario from previous tests with with long-resume unblocking, generated anticipated results 3.17, with increased network performance (to about 157 MBit/s). Note that the received CPU-times for the domUs are offset by the same amount, rather than scaled, which is due to the uniformly

distributed top-up (compare to figure 3.10). Increased demand of dom0 can be seen in the drop of the received CPU-time during I/O activity.

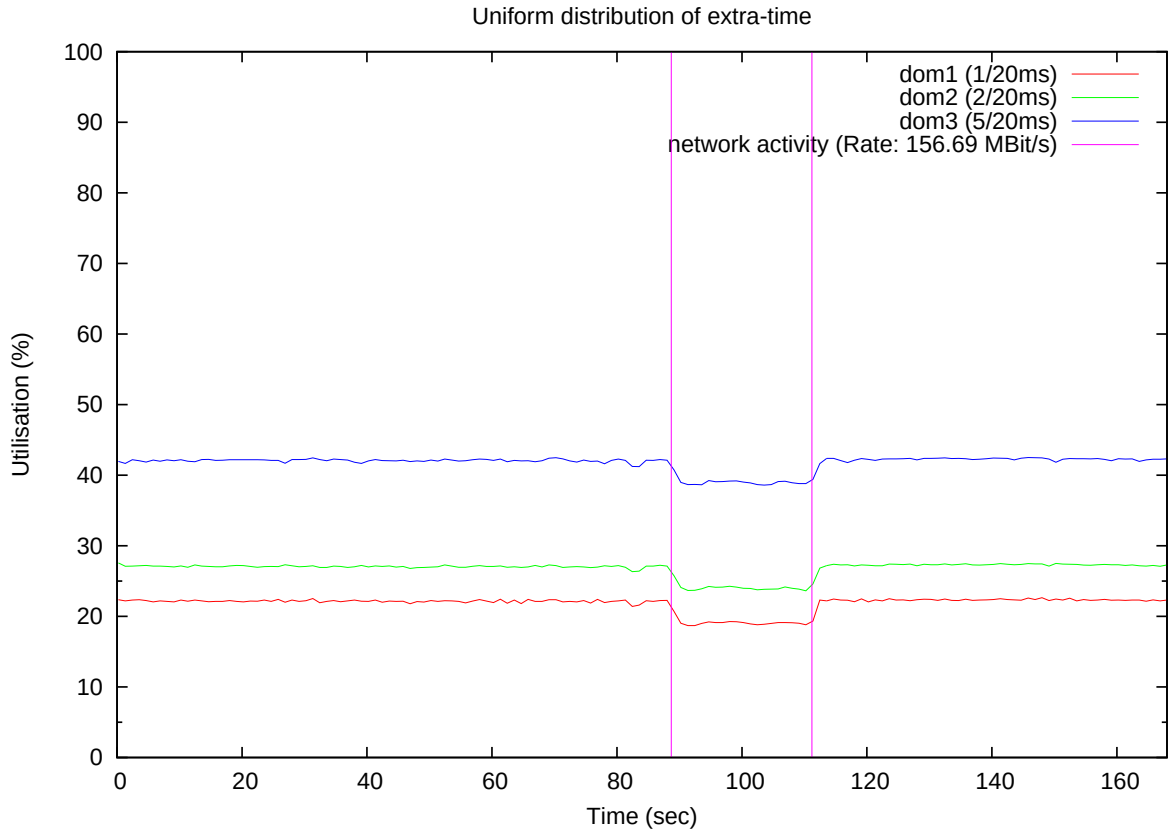


Figure 3.17: Distributing idle time in the system among the domUs in uniform round-robin fashion.

**Weighted distribution** Distributing CPU-time uniformly among VCPUs present in the system is not ideal if the system's administrator wants to set ratios between the VCPUs as it does not take unequal guarantees into account. Given that, there is a need for weighted distribution of slack time in the system.

In order to implement weighted extra-time distribution, I introduced a score $sc_i$ for each VCPU, which would serve as a key for sorting VCPUs that are marked extra-aware in ascending order on the extra-queue. A slack time quantum of size $e$ is given to the first VCPU (the one with the lowest score). After it has finished this quantum, its score is increased by some factor of the inverted weight $w_i$, given by the administrator or by some factor of the inverted utilisation $p_i/s_i$ of the VCPU $i$.

That is

$$sc_i' = sc_i + \begin{cases} \alpha_1 \cdot {w_i}^{-1} & \text{if } w_i > 0 \\ \alpha_2 \cdot p_i/s_i & \text{otherwise} \end{cases} \tag{3.3}$$

It plain to see that VCPUs will then receive slack time in ratios according to their specified weights or given guarantees. Let $e_{recv,i}$ be the extra-time received by VCPU $i$ so far. Then $e_{recv,i}/e_{recv,j} = w_i/w_j$

for all $i, j \in V_w$, with $V_w$ the set of all extra-aware VCPUs with specified weight, similarly

$$\frac{e_{recv,i}}{e_{recv,j}} = \frac{s_i/p_i}{s_j/p_j}$$

for all $i, j \in V_n$, with $V_n$ the set of all extra-aware VCPUs without specified weight.

Scaling $\alpha_1$ and $\alpha_2$ in equation 3.3 does not change the outcome of the scheduling algorithm, however in order to speed up comparison and calculation, fixed point arithmetic with ten fractional bits is used, setting $\alpha_2$ to $2^{10} = 1024$. Setting $\alpha_1$ then determines the ratio between the amount of extra-time a VCPU with set weight receives and the amount of time, a VCPU without a weight can consume.

For flexibility, $\alpha_1$ is set to $\alpha_1 = 128 \cdot \alpha 2$, resulting in the fact that a VCPU $i$ with weight $w_i = 64$ does receive the same amount of extra-time as a VCPU $j$ with a utilisation of 50% , *i.e.*, $s_j/p_j = 0.5$, allowing to adjust the balance between explicit and implicit weights.

Adding to each VCPUs score $sc$ eventually causes a numerical overflow. To prevent this, subtracting a constant from the score $sc$ of all VCPUs can be done, either once an overflow is likely to occur, or from time to time. As the sorted extra-queue is represented as a linked list, one can also do the following: Instead of adding the score increment $si_i$ to the score $sc_i$ of VCPU $i$, one can also subtract $sc_i$ from all scores $sc_j$ of the other VCPUs $j$, where $i \neq j$ and simply set the new score of VCPU $i$ to the increment: $sc_i' = si_i$. This avoids overflows, as the scores of the VCPUs always stay around zero. The cost is a full traversal of the queue when re-enqueueing a VCPU into the extra-queue, instead of the half traversal, which is to be paid on sorted enqueuing on average.

The algorithm works as expected, as can be seen from the results in figure 3.18, maintaining the same *ratio* among the received CPU-times, rather than increasing them by a constant offset (again, please compare to figure 3.10), even when less slack time is available during the I/O interval. In comparison to 3.17 the network performance has increased further to about 264 MBit/s, which can be explained by an increased amount of extra-time being available to dom0, because of its scheduling parameters of 10/20 ms.

**Helping I/O-bound domains** In subsection 3.2.3 we looked at several ways to improve service for VCPUs that were I/O-bound. Various ways of modifying the real-time scheduling parameters have been examined, however, it was found that just bringing down latency to an absolute minimum did not help all applications, which is due to the asynchronous buffering effect caused by slow unblocking.

Additionally, penalties occured, because of frequent context switches and the associated overheads. Not penalising other VCPUs by simply not switching to them during phases of heavy I/O as long as possible is possible, and would mean to push back contracted execution of VCPUs back as far as possible. This is

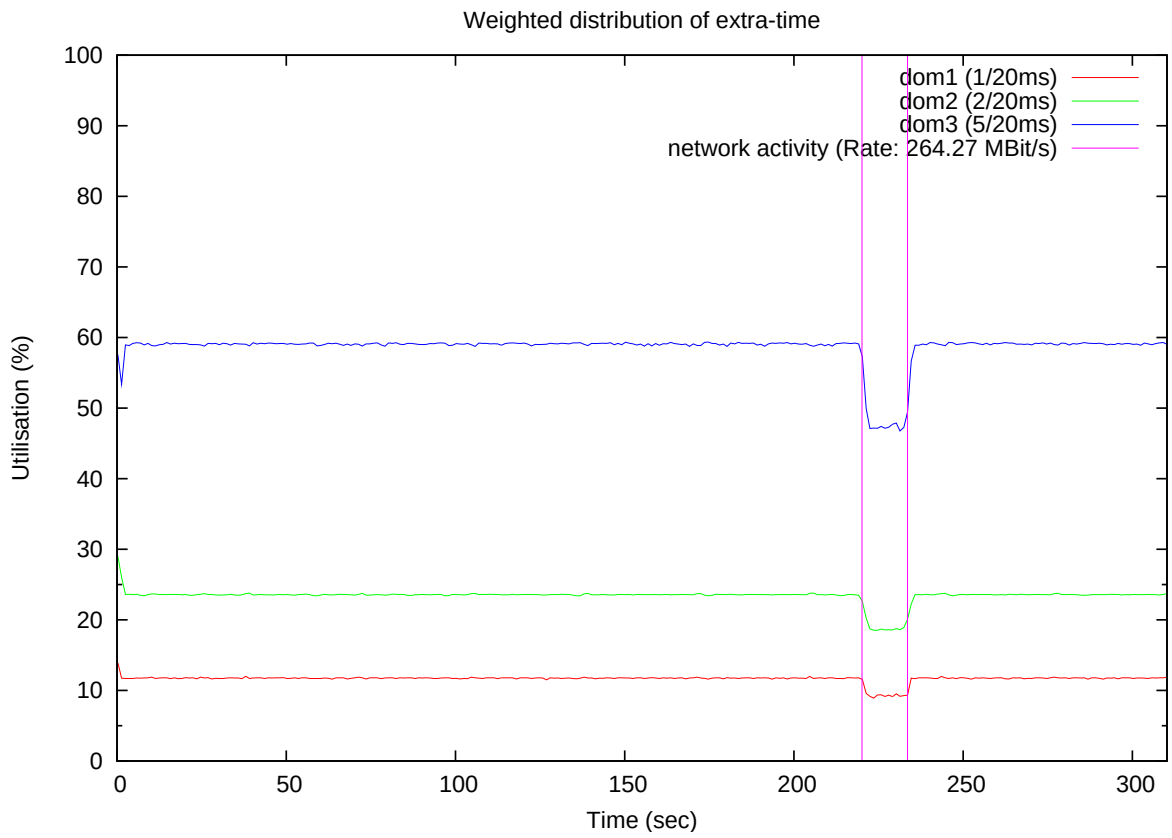


Figure 3.18: Distributing idle time in the system proportionally to slice/period ratio.

done in *slack-stealing* approaches [Liu00f], but as it incorporates the nontrivial taks of estimating slack-time, this has not yet been implemented. Another possibility would be to use *least slack time* scheduling (see [MD78] for an introduction) straightaway.

But staying with the established EDF behaviour, the question remains: When it is useful to give time to I/O-bound VCPUs? There is a plain answer to this, it is safe and reasonable to give additional time to I/O-bound domains, once all commitments have been fulfilled (for now) and the cpu is idle.

As a VCPU that blocks and thus gives away parts of its time-slice creates a subsequent hole of slack time later in the schedule, other VCPUs with guarantees can receive their time slice earlier. Given that the resulting slack time present in the system was caused by a blocking domain, it is just fair to give it back to this VCPU preferentially, which is similar to the idea of *backgrounding* servers for aperiodic jobs [Liu00g].

In the simple round-robin approach of distributing extra-time, simply putting the unblocking VCPU to the head of the extra-queue (instead of the back) already boosts the performance of I/O-bound domains. But as distributing extra-time in round-robin fashion is not too useful, we will have a closer look at the weighted approach of extra-time distribution.

**Multilevel slack time distribution** Weighted distribution of slack time allows us to differentiate between domains which have suffered more and those who have suffered less from penalties due to unblocking policy. The idea is to have some metric $m_{pen}$ on this penalty and then distribute slack time weighing each VCPU with this metric. Additionally, VCPUs that apply for compensation in this way, should be granted slack time before all VCPUs that are just flagged as extra-aware.

This leads to the idea of a second level extra-queue, which has a higher priority than the normal extra-queue. This new queue is referred to as *penalty-queue*, as it serves the purpose to support penalised VCPUs. Given this new queue for slack times, the scheduler is now managing a total of four queues, which are the runnable-queue, the waiting-queue, the standard extra-queue and the penalty-queue. The next paragraph will summarise all queues and their functions.

The metric $m_{pen}$ that I have mentioned should somehow capture the loss which has occurred to a VCPU $v_i$ and relate it to other losses suffered by other VCPUs. What is actually a loss for a VCPU, that has deliberately done I/O and then blocked on a resource? In terms of the aforementioned blocking nomenclature, a loss actually happens, if a VCPU can not consume its full slice during a period due to unblocking policy. Given that, there is no loss for long-unblocking, as we can simply shift the periods, so that no time is lost for the VCPU. Loss can only occur when a VCPU does a short-unblock and then does not get the CPU again in this period, because otherwise guarantees for other VCPUs would be in danger (see subsection 3.2.3 for details). Availability of slack time later on then allows us to give back some of the time that was originally lost for the VCPU.

The penalty metric used in my experiments is computed similar to the CPU-utilisation of a VCPU:

$$m_{pen,i} = \frac{t_{pen}}{p_i} = \frac{s_i - c_i}{p_i} \tag{3.4}$$

where $m_{pen,i}$ is the penalty metric for VCPU $v_i$, $s_i$ and $p_i$ denote the usual guarantee-parameters (slice and period accordingly), $c_i$ is simply the CPU-time already consumed by the VCPU during the current period.

Once a VCPU unblocks during the same period it blocked, the metric is computed as above and then used similar to the implicit weight in the normal extra queue, *i.e.*, it's inverse is used as the VCPU's score when enqueuing it into the the sorted penalty-queue:

$$sc_{pen,i} = \alpha_3 \cdot \frac{p_i}{s_i - c_i} \tag{3.5}$$

Note that during this operation, we do not increment the score after a VCPU has received slack time for penalty compensation, as due to the increased CPU-time $c_i$, the score is increased automatically. Again, setting $\alpha_3$ does not change the order in the penalty queue. Similar to above I chose fixed-point calculation

with 10 fractional bits and thus $\alpha_3 = 2^{10} = 1024$. Note also, that the ratio $\alpha_1/\alpha_3$ (and $\alpha_2/\alpha_3$ likewise) are irrelevant, as the scores are used on different queues. Computation of the penalty also ensures that a VCPU does not receive more CPU-time than contracted in the case of compensation for block-loss. If additional slack-time in the system is to be given to the VCPU, it needs to request it from the

Unblocking with the support of an additional extra queue is refered to as *extra-support unblocking*.

Testing (again, standard setup) of this supporting unblocking scheme resulted in good results in terms of network performance *and* the quality of guarantees provided to other domains. Figure 3.19 shows the results when extra-time is just distributed for I/O-support, keeping received CPU-time for the domUs at the contracted levels. Figure 3.20 shows the results for full two-level extra-time distribution, as remaining extra-time (after compensation) is given to the VCPUs of extra-aware domains. The network performance is higher than in BVT (see figure 3.7) in both cases and guarantees are still managed correctly. Extra-time still is distributed according to the slice/period ratio, also during the presence of I/O phases and another level of extra-time distribution.

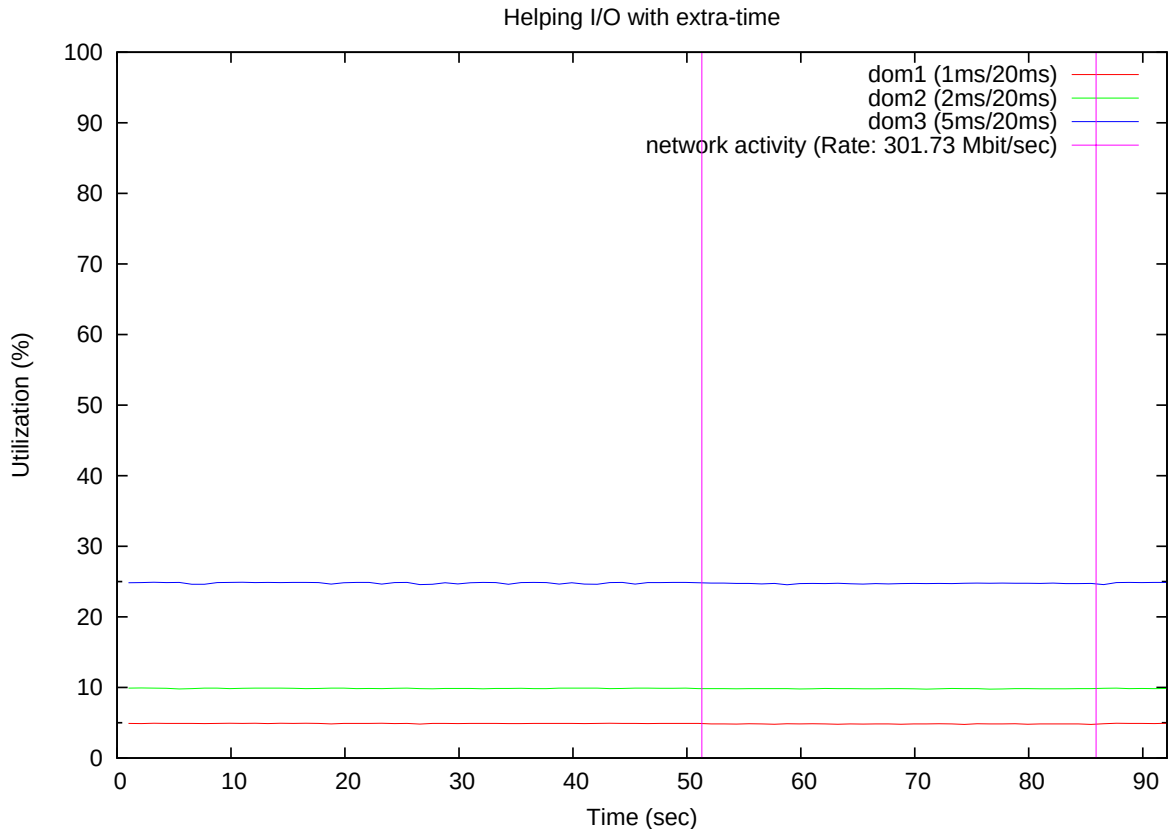


Figure 3.19: Using idle-time in the system to support I/O driven domains.

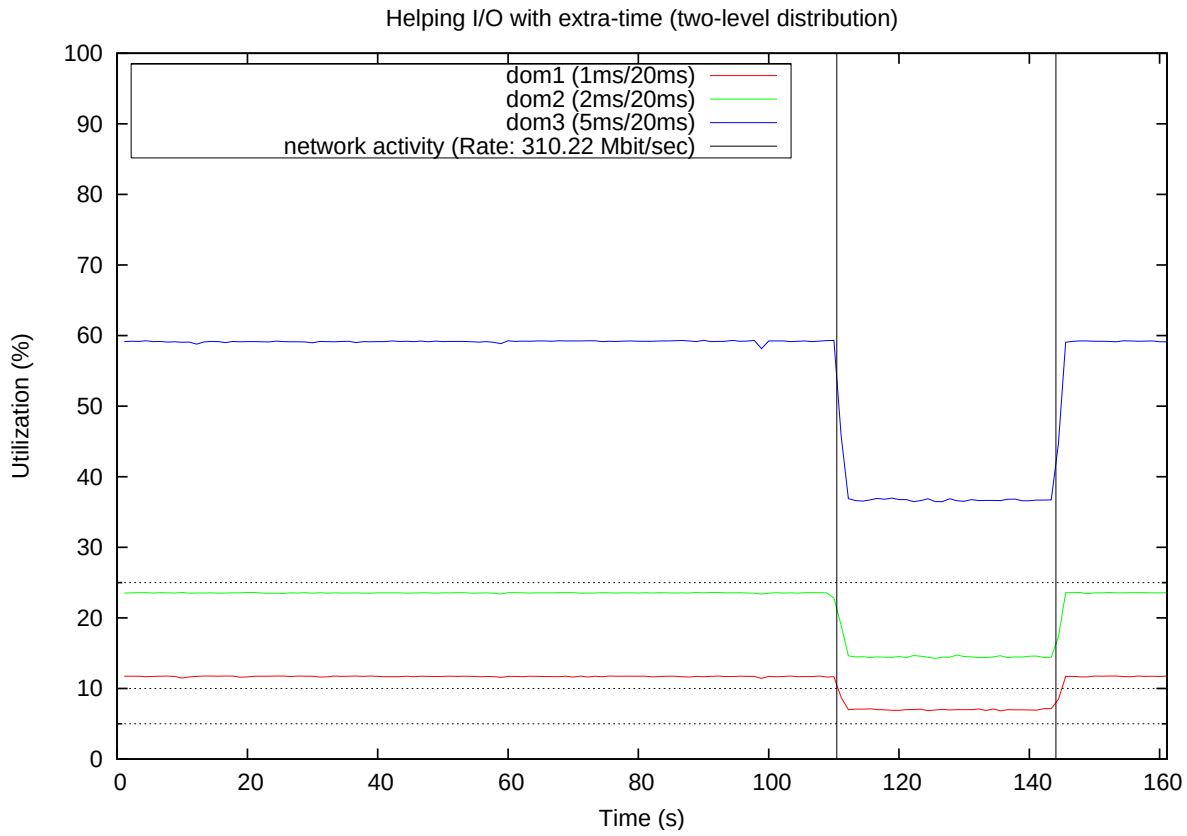


Figure 3.20: Using idle-time in the system to support I/O driven domains, giving extant idle-time to domUs. Guarantees are enforced also during I/O, as can bee seen by the dashed lines that correspond to the given guarantees.

### 3.2.5 Architectural summary

**Queues** Given the tweaks introduced in the previous subsection, the final SEDF scheduler consists of four queues: the runnable-queue, the waiting-queue, the penalty-queue and the extra-queue, see table 3.1 for an overview, about what is contained on each queue, how they are sorted and how scheduling decisions are made.

| Queue | Priority | Sorted by | VCPUs contained |
|---|---|---|---|
| runnable | 3 | deadline | runnable, non-empty slice for current period |
| waiting | - | start of next period | empty slice, awaiting next period |
| penalty | 2 | inverse of penalty metric | penalised by short-unblocking policy |
| extra | 1 | inverse of weight (implicit / explicit) | extra-aware |

Table 3.1: Queues, present in the single-cpu SEDF scheduler.

Given these queues, finding the next VCPU to schedule then just comprises to check the queues in descending priority order and to select the head of the first non-empty queue encountered. If all queues (apart from the waiting queue) are empty, the idle-VCPU is selected to run.

**Types of supported domains** With the introduced slack-time distribution, the scheduler not only supports domains with strict guarantees, but also handles VCPUs running without any guarantees, called *best-effort domains*. Domains that use both, *i.e.*, partly guaranteed execution and additional slack time can also exist in the system and are called *mixed-mode domains*.

The user / system administrator can specify the scheduling parameters for all VCPUs of a domain, either by specifying a pair of slice and period values and an optional extra-time-awareness flag or by just giving a weight to a domain directly. When a weight for the specification is used, it can either be used for a guarantee or directly as a weight for a best-effort domain, as described above.

When the given weight is intended to be used for a guarantee, the SEDF scheduler converts this weight into a suitable pair of slice and period lengths, by looking at the guarantees handed out already to domains with specified slice/period-pairs, estimating the remaining CPU-time that can be guaranteed to real-time domains with weight-specification. The available remaining CPU-time is then divided according to the specified weight, creating suitable slices and period lengths, thereby avoiding overload of the system

(given that the specified slice/period-VCPUs leave some spare CPU-time).

More precisely, a VCPU $v_i$ is assigned a share of $s_i/p_i$ from a given weight $w_i$ by

$$\frac{s_i}{p_i} = \frac{w_i}{w_{total}} \cdot \left(1 - \sum_{k \in F} \frac{s_k}{p_k}\right)$$

where

$$w_{total} = \sum_{j \in W, j \neq i} w_j$$

and $F$ the set of VCPUs with user-specified slices and periods and $W$ the set of VCPUs with weight-defined guarantees.

## 3.3 Results

During the last section, I have given results for various micro-benchmarks highlighting effects caused by various algorithmic ideas. In this section I want to show SEDF's performance under more real world workloads, namely a compilation of the Linux kernel and a small Apache benchmark. These two benchmarks are by no means extensive, but they give an indication of what to expect from SEDF's performance and how scheduling parameters translate into compile times and web server performance, respectively.

### 3.3.1 Linux kernel compile

In this benchmark, I compile a standard Linux kernel on the single CPU of my test machine (same as above, single 2.4 GHz Xeon, 2 GB RAM), running Xen with various scheduling settings. The Linux kernel of version 2.6.12 is compiled with the command `make -j2`, which is commonly used on a single core CPU. All files needed for and created during the compilation rest on a file server, which is connected to the test machine over Gigabit-ethernet.

The scheduler operates in the "two-level extra-time" mode, which has been the most efficient mode in the earlier experiments which would also honour guarantees. Initially, the compilation takes place in dom0, with slice/period set to 20ms each, giving dom0 full access to CPU-time. This test should result in the fastest possible compilation.

The compilation process is then repeated in unprivileged domains with various settings, initially just in one of them and then the compilations are carried out concurrently in three VCPUs with the scheduling settings 1ms/20ms, 2ms/20ms and 5ms/20ms. All VCPUs are *not* flagged to recevie slack-time once they have made up for their blocking penalty, ensuring that they do not receive more CPU-time than guaranteed.

Note that once a compilation was finished in one of the faster domUs, it was restarted (with `make clean` in between of course) in order to not let the CPU become idle. Table 3.2 lists the averaged[5] resulting compilation times and the resulting ratios according to scheduling parameters for the single compilations. Results for concurrent compilations can be found in table 3.3, showing the results in the three domUs. Figure 3.21 compares the results graphically.

| slice/period | Avg. compile time | Ratio to dom0 | Ratio expected |
|---|---:|---:|---:|
| 20ms/20ms* | 737,13 s | 1 | 1 |
| 1ms/20ms | 12612,37 s | 17,11 | 20 |
| 2ms/20ms | 6166,80 s | 8,37 | 10 |
| 5ms/20ms | 2498,26 s | 3,39 | 4 |

Table 3.2: Compilation of a Linux kernel with various scheduling settings in an unprivileged domain. (Compilation marked with * was run in dom0.)

| slice/period | Avg. compile time | Ratio to dom0 | Ratio expected |
|---|---:|---:|---:|
| 20ms/20ms* | 737,13 s | 1 | 1 |
| 1ms/20ms | 15238,73 s | 20,67 | 20 |
| 2ms/20ms | 6959,43 s | 9,44 | 10 |
| 5ms/20ms | 2725,08 s | 3,70 | 4 |

Table 3.3: Simultaneous compilation of a Linux kernel in different domUs with various scheduling settings. (Compilation marked with * was run beforehand in dom0.)

As can be seen from the results, scheduling parameters translate very well to measured performance, with a benefit for domains with less contracted CPU-time, when running alone. Most likely this occurs, because access times to the data on the network storage do not increase proportionally to the reduction of the domUs slice length, giving the domain a better overall performance. Which is in order, as were are giving guarantees about received cpu-time and cannot influence seek times on the network attached storage.

When multiple compilations are run simultaneously, performance still stays predictable, even though concurrent access to the storage system occurs and slows down compilation.

[5] for raw data see appendix A.1

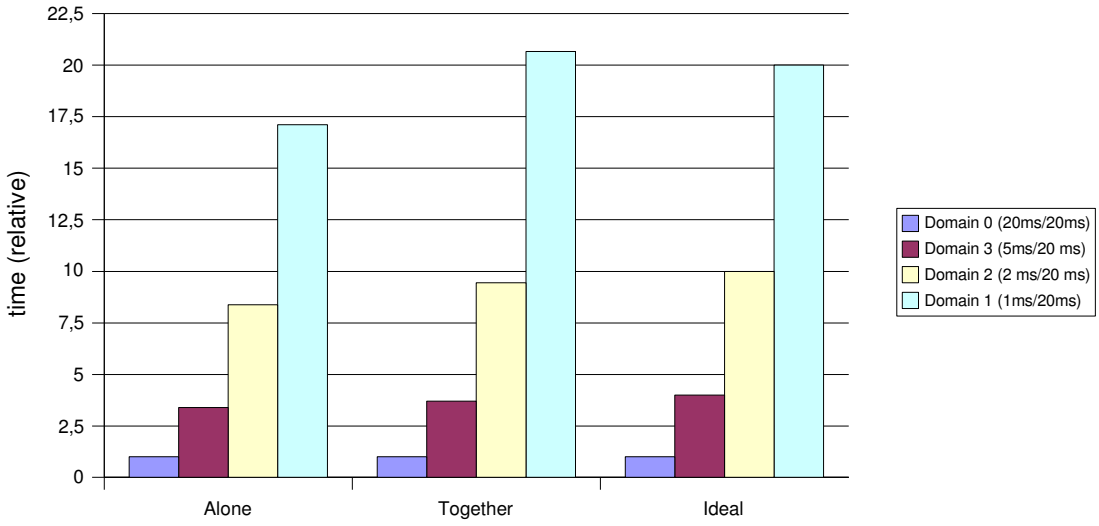


Figure 3.21: Compile times for a Linux kernel in domains with different scheduling settings, relative to performance in dom0 with 100% CPU-time. (Lower bar is faster.)

### 3.3.2 Apache web server

As one of Xen's common domains is server consolidation [Xen], server performance is an important measure. This test uses the well-known Apache web server [Apa], and measures its performance with the included "ab" tool, which requests a certain web page (Apache's default web page in our test) and measures the amount of fulfilled requests per second. In order to simulate many clients, "ab" issues 100 concurrent requests, instead of waiting for each request to finish, before issuing the next one.

The benchmarking tool runs on a different (physical) machine than the server, both machines are connected over a Gigabit-ethernet. The web server runs on the standard test machine (2.4 GHz Xeon, 2 GB RAM) inside a domU in Xen. Two tests are conducted, in the first, the CPU-share of the domU, which runs the web server, is increased and the effect on the web server's throughput is measured. The second test analyses the impact of dom0's CPU-share on the web server's performance. For that, domU receives constant service (10ms/20ms) and the guarantees for dom0 are reduced.

Changing the scheduling settings translates very well to perceivable performance on the service level, as can be seen in the results from the first test in figure 3.22.

Reducing the CPU-share of dom0 starts to affect the web server's performance slightly at shares below 15%, showing that there is an extremely small I/O-overhead added by domain 0. The drop in performance at a share of 45% for dom0 is not related to scheduling issues, but likely is due to external issues, such as network congestion, see the appendix for raw measurement data A.2. Appart from this anomaly, the I/O domains share does not influence the web servers performance much, which shows that it is not the

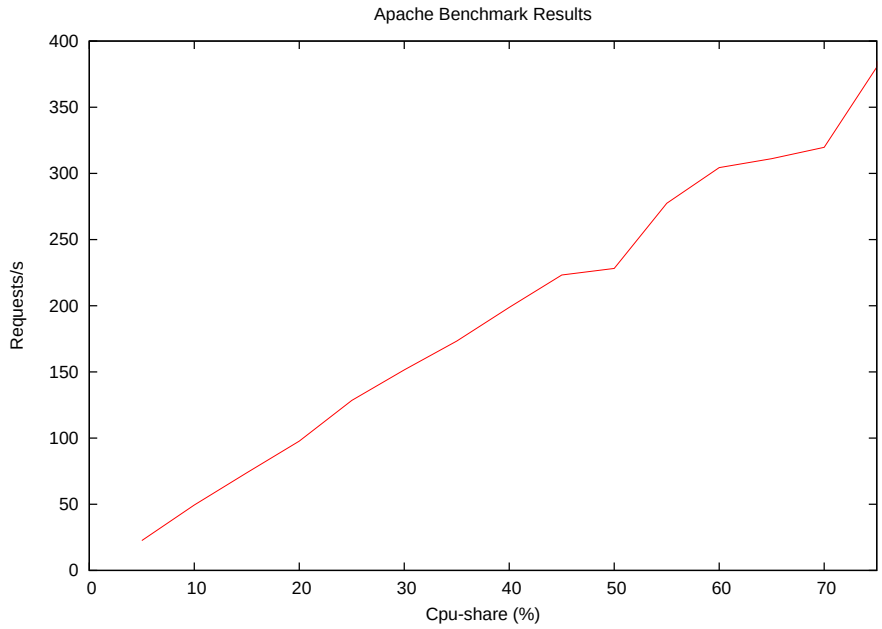


Figure 3.22: Relationship between domU's scheduling parameters and number of processed requests in Apache.

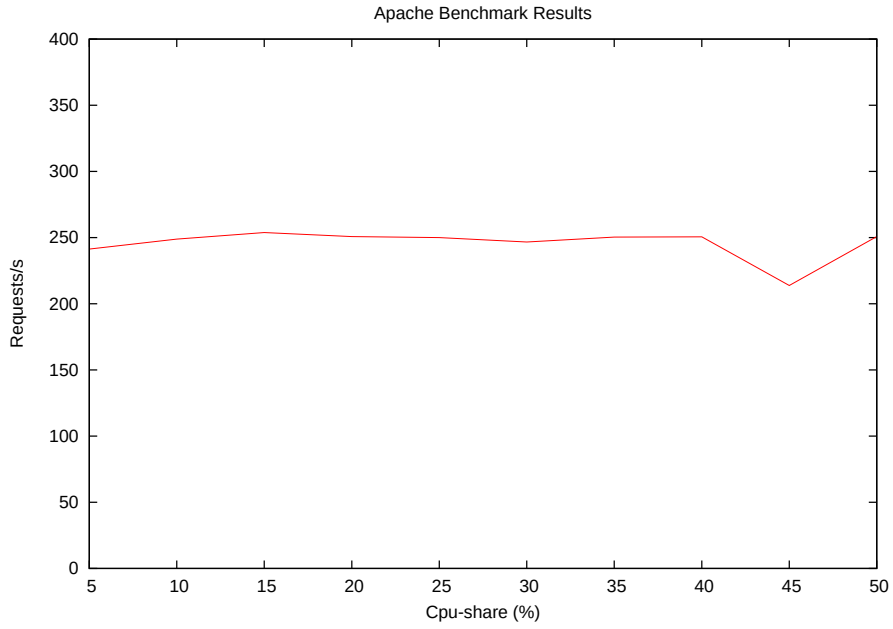


Figure 3.23: Relationship between dom0's scheduling parameters and number of processed requests in Apache.

bottleneck in the first setup.

In order to rule out bottlenecks on the client side (where “ab” is run), a third test repeats the first test, mentioned above, but with two client machines. Figure 3.24 shows the bandwidth for each client and the sum of both. Results are identical to figure 3.22, which is a good indication that the client has not been the bottleneck and that provided service does not degrade if multiple machines use the web server.

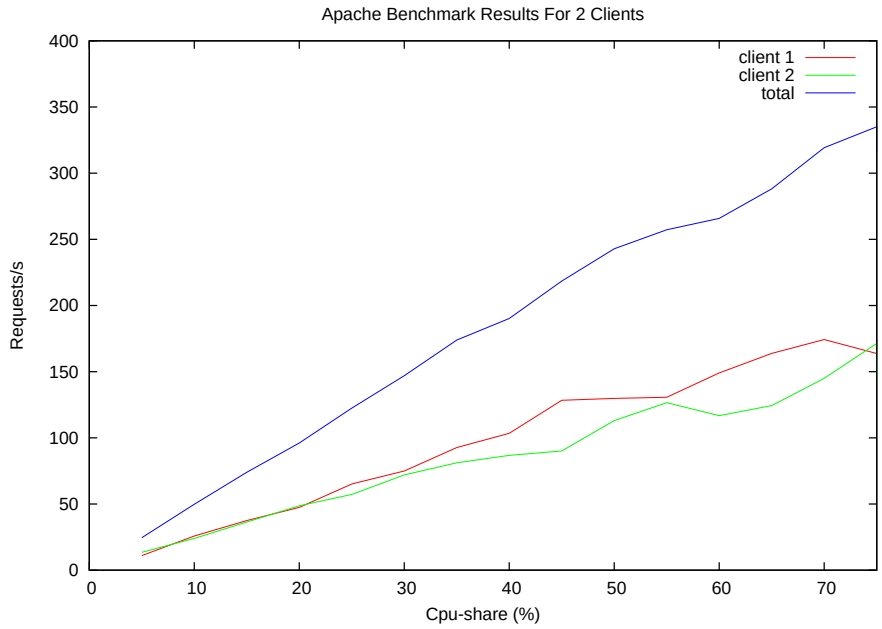


Figure 3.24: Two clients sending requests to Apache.

## 3.4 Summary

### 3.4.1 Conclusion

This chapter introduced a new scheduler SEDF for the Xen hypervisor. The scheduler is capable of providing firm guarantees to virtualised guest operating-systems on a single processor. Additionally, it supports best-effort execution of guests by distributing unused, available time in a weighted fashion among them. This feature set is supported by excellent performance, due to a novel and sophisticated treatment of I/O, exceeding Xen’s previous default scheduler’s (BVT) performance, while at the same time having a larger set of features to control guest execution.

It has also been shown that the raw CPU-time given to the guest translates well into overall application performance, this has been demonstrated with experiments using Apache as a standard web server and a Linux-kernel compilation. However, more testing results of more complex workloads, such as in real-live scenarios would be very helpful.

### 3.4.2 Outlook

At the time of writing, the SEDF scheduler, developed in early 2005, is still present in Xen, but has been replaced as default by a ticket-based scheduler, which has not yet been investigated into by the author. SEDF still requires some work, especially minor code cleanup, testing and incorporation of results gathered by real-world usage and general bug-fixing. However, as the author has been involved a lot with his course of studies, the scheduler is pretty much unmaintained at the moment.

An additional interesting point, which has not yet been investigated, is the question of whether we can really rely on the loose, asynchronous coupling between domains. Especially for small I/O transfers, that occur frequently but need immediate replies before the next request can be issued, might be better handled with synchronous communication primitives and according scheduling policies*e.g.* as found in [SWH05]. With unfortunate scheduling policies (for example isochronous unblocking in SEDF) performance may drop excessively, as perceived by the author when booting a domU from an NFS root (this took about 20 min with the isochronous SEDF setting). Using the improvements mentioned in this chapter, this improved tremendeously, however, a more detailed analysis of these scenarios is necessary, especially with the upcoming driver-domains that will exhibit a strong I/O-driven and synchronous behaviour.

# 4 N:M scheduling on multiple processors

## 4.1 Introduction

This chapter describes the operation of the (experimental) balancer of Xen, written by the author. The balancer was developed during a one-year visiting research course at the University of Cambridge's Computer Lab in 2004/2005. Most of the work was done during spring and summer 2005, with preceding works on Xen's new scheduler SEDF, standing for Smart / Simple Earliest Deadline First scheduler, which provides domains with guarantees of CPU share. It is however extended to provide best-effort service, too (see 3), and it is this functionality that the balancer addresses, as it provides fair balancing for best-effort domains.

## 4.2 Algorithm

### 4.2.1 Overview

The balancer is a deterministic global balancer. It tries to balance system load by using the current state and then migrating domains between CPUs, *i.e.*, it works incrementally. Strict convergence has not been proven, and is not wanted in some particular corner cases, which will be discussed later. Divergence is avoided by calculating ideal amounts of domains to be moved.

Once the balancer is called, it gathers information on the state of the CPUs and updates also domain data. The data collected is mainly timing data (received time, available time) and weight (domains have a weight which affects the amount of time they receive). During this step the balancer evaluates the "cpumap"-flags for each VCPU, determining the potential balancing groups of CPUs and creating one or more groups of equivalency, isolated "islands" of CPUs that cannot balance to another "island". See figures 4.1 to 4.3 for examples of such groups. These groups are the basic entities for the scheduler, as no matter how hard it tries, it cannot balance between those groups.

CPUs in each group are sorted by their load and then load is balanced pairwise between them, starting with the combination highest load - lowest load, and then 2nd highest - 2nd lowest load, *etc.* Of course

the information about possible target CPUs is taken into account, so it is made sure that $cpu_{low-load} \in targets(cpu_{high-load})$, as this is not always the case in some balancing groups, further details can be found in a later subsection (4.2.3).

Once a suitable pair of CPUs has been selected, the ideal amount of weight $w_{ideal}$ that should get moved from the highly loaded (hot) CPU to the one with lower load (idle) is calculated. This value is ideal and would create a perfect equilibrium of load between those two CPUs. Trying to get the actual value of the total weight $w_{total}$ moved as close as possible to this value is therefore desirable. $w_{total}$ is simply the sum of all the weight that gets moved from the hot to the idle CPU minus the sum of the weight, that gets moved in the opposite direction. The latter action seems to be counter-intuitive, useful application of this will be discussed shortly. It is quite obvious that trying to get $w_{total} = w_{ideal}$ is an instance of the knapsack (or subset sum) problem, which is proven to be NP-complete in [Sip97]. The current balancer therefore uses an poly-phase algorithm, where the first and second phase posses worst-case execution time (wcet) of $O(n)$ and the third phase is $O(n^2)$ on the given data structures (double linked lists). If an early phase finds a $w_{total}$, that is close enough to $w_{ideal}$, the following phases are not executed.

$W_{ideal}$ gives a quantitative hint about the domains to move, but there is still the choice, which domains should be moved, *i.e.*, a qualitative hint. This hint is delivered by the history of each domain, more specifically its received CPU-time so far. The balancer moves domains that received less time than they are entitled to, from the hot to the idle CPU and those, which received more in time in the past, are moved onto the hot CPU. This however relies on the fact, that we do not overcompensate, *i.e.*, after the balance the hot CPU does not turn into an idle cpu. But this can only happen, if we move to much weight, and it is therefore avoided by staying close to $w_{ideal}$.

Once the swap-set for this pair of CPUs is determined, the migration is carried out and balancing continues with the next pair of CPUs, where (of course) the just considered CPUs are not taken into account as migration sources / targets.

In order to avoid unnecessary (and timely) runs through the balancer, thresholds are incorporated that specify whether it is reasonable to perform the intended balancing operation. In the selection of pairs of CPUs for balancing, the load difference needs to exceed a certain value, otherwise balancing is not performed.

### 4.2.2 Balancing details

**The task model** The balancer deals with VCPUs that are scheduled in *best-effort* fashion by the underlying real-time scheduler. Idle CPU-time that is not used by any of the real-time VCPUs (either

because of underutilisation, I/O blocking or deliberate release of guaranteed time) is spread among those VCPUs that are declared aware of extra-time. VCPUs that rely solely on this time for their own execution are called *best-effort VCPUs* (BE), those that additionally have a guaranteed time of execution are named *mixed-mode VCPUs* (MX).

The balancer deals with be-VCPUs only, it does not touch rt- or mx-VCPUs, as the balancing is likely to interfere with the timing guarantees and would need an extra-admission control, which is implemented in user(-space). The underlying scheduler distributes extra time on each CPU to be-VCPUs on that cpu proportional to their weight.

Let $w_{v,i}$ be the weight of be-VCPU $v_i$, $w_{cpu,k}$ the total weight of all be-VCPUs on CPU $k$, $bedoms_{cpu,k}$ the set of all be-VCPUs on CPU $k$, $t_{recv,i}$ the CPU-time the be-VCPU $v_i$ received in the last interval $\Delta t$, $t_{avail,k}$ the time available in the last interval for non-rt VCPUs on CPU $k$ and $t_{rt,k}$ the total amount of time consumed by guaranteed execution on CPU $k$ in the last interval.

Then

$$w_{cpu,k} = \sum_{v_j \in bedoms_{cpu,k}} w_{v,i}\text{(trivial)}$$

$$\frac{t_{recv,i}}{w_{v,i}} = \frac{t_{recv,j}}{w_{v,j}}$$

for $v_i$ and $v_j$ on the same CPU (guaranteed by underlying scheduler).

$$t_{avail,k} = \Delta t - t_{rt,k}$$

and

$$\sum_{v_i \in bedoms_{cpu,k}} t_{recv,i} = t_{avail,k}$$

$$t_{recv,i} = w_{v,i} \cdot \frac{t_{avail,k}}{w_{cpu,k}}$$

where $v_i \in bedoms_{cpu,k}$.

**Main balancing principle** Let $\tau_{v,i}$ be the received CPU-time per weight for VCPU $v_i$, *i.e.*, $t_{recv,i}/w_{v,i} = \tau_{v,i}$. The underlying scheduler should then ensure for each CPU $k$: $\tau_{v,i} = \tau_{v,j}$ for all $v_i, v_j \in bedoms_{cpu,k}$. We then also have a $\tau_{cpu,k}$, which is the CPU-wide received time per weight, which is $\tau_{cpu,k} = \tau_{v,i} = t_{avail,k}/w_{cpu,k}$. The main task of the balancer is to ensure that $\tau_{v,i} \approx \tau_{v,j}$ for $v_i$ and $v_j$ on different CPUs $k$ and $l$. As we have seen this can be achieved by trying to make $\tau_{cpu,k} = \tau_{cpu,l}$.

This is achieved by moving domains from one CPU to another. Let us have a look at what happens in such a case, as VCPU $v_i$ is migrated from CPU $k$ to CPU $l$ (let $x'$ denote the value of parameter $x$ after

the migration): Obviously

$$w'_{cpu,k} = w_{cpu,k} - w_{v,i} \tag{4.1}$$

$$w'_{cpu,l} = w_{cpu,l} + w_{v,i} \tag{4.2}$$

$$\tau'_{cpu,k} = t'_{avail,k}/w'_{cpu,k} \tag{4.3}$$

$$\tau'_{cpu,l} = t'{}_{avail,l}/w'_{cpu,l} \tag{4.4}$$

As established, it is the aim to get $\tau'_{cpu,k} = \tau'_{cpu,l}$ in order to get $\tau_{v,i} \approx \tau_{v,j}$ with $v_i \in bedoms_{cpu,k}$ and $v_j \in bedoms_{cpu,l}$. In order to achieve that, we need to find a suitable weighted domain $v_i$. After some algebra we get

$$w_{v,i} = \frac{t'_{avail,l} \cdot w'_{cpu,k} - t'_{avail,k} \cdot w'_{cpu,l}}{t'_{avail,k} + t'_{avail,l}} = w_{ideal}$$

assuming that $t'_{avail,k} = t_{avail,k}$ and $t'_{avail,l} = t_{avail,l}$.

That means in order to satisfy $\tau'_{cpu,k} = \tau'_{cpu,l}$, we need to move a VCPU $v_i$ with $w_{v,i} = w_{ideal}$ from CPU $k$ to CPU $l$. The same argument holds, if we do not move a single VCPU $v_i$, but rather a set of VCPUs, whose total weight equals $w_{ideal}$.

In a real-world scenario, the situation is slightly more complex, due to increased dynamics in the system. As an example $t_{avail}$ does not stay constant over time for a given CPU, as real-time domains may behave differently or get added to/removed from the system. Therefore it is necessary to do the balancing regularly. Additionally, $w_{ideal}$ in this case is an abstract value, and it may be impossible to comply exactly in practice. This can be demonstrated with a simple example:

Given: Two equal CPUs, which are both completely idle, and one be-VCPU $v_1$, weight $w_{v,1} = 1$. Assume the be-VCPU is located on CPU $k$ first. It is obvious that $t_{avail,k} = t_{avail,l} = t_{avail}$ and $w_{ideal} = (t_{avail} \cdot 1 - t_{avail} \cdot 0)/(2t_{avail}) = 0.5$

However, moving half of VCPU $v_1$ is (unfortunately) not possible, as a VCPU does not bear any exploitable (at least to Xen) parallelism in itself. Therefore we can either move the domain, or not. One might argue, that in this case it does not matter, because the domain will receive equal service (from our point of view) on either CPU, so a move wouldn't do any good, so it should be avoided.

But in more complex cases we have to accept the fact, that no distribution can be ideal, so we cannot ensure that $\tau_{v,i} = \tau_{v,j}$ for all $v_i, v_j$. We would rather like to make sure, that each VCPU receives equal service over time , *i.e.*, $\tau_{v,i}$ is evened over a longer time span. Therefore we use a simple and effective recursive filter to average $\tau$ over time: $\tilde{\tau}' = 1/2 \cdot \tilde{\tau} + 1/2 \cdot \tau'$

Relaxing the demands on the balancer in this way allows us to tackle the following prevalent problematic case:

We have again two equal CPUs $k$ and $l$, both idle (without any rt-VCPUs) and three be-VCPUs, each with weight 1. Let the initial configuration be $bedoms_{cpu,k} = \{v_1, v_2\}$ and $bedoms_{cpu,l} = \{v_3\}$. Trying to transfer from CPU $k$ to $l$, we find the ideal weight to be $w_{ideal} = 0.5$.

Contrary to above, we cannot just ignore the problem and do nothing, as both $v_1$ and $v_2$ receive only half of the amount of CPU-time as $v_3$ does. Hence $\tau_{v,1} = \tau_{v,2} = 1/2 \cdot \tau_{v,3}$. If we introduce the notion of

$$\tau_{ideal} = \frac{\sum_{k \in CPUs} t_{avail,k}}{\sum_{k \in CPUs} w_{cpu,k}}$$

we find that $\tau_{ideal} = 2 \cdot t_{avail}/3 = 2/3 \cdot \tau_{v,3} = 4/3 \cdot \tau_{v,1} = 4/3 \cdot \tau_{v,2}$. Making it obvious again, that VCPU $v_3$ is getting more CPU-time than it actually deserves.

If we accept imbalances in the current $\tau_{v,i}$ and try to balance the averaged case, we are able to do the following: Allow each VCPU $v_1$, $v_2$ and $v_3$ a "place in the sun", *i.e.*, to run alone on the CPU for a while and take turns in doing so. This can either be achieved by moving one domain from the hot to the idle CPU, therefore changing roles (the hot CPU now becomes the idle one) and giving the benefits of an exclusive CPU to the domain staying behind. This would mean to exceed $w_{ideal}$ ($w_{moved} = 1$ and $w_{ideal} = 0.5$) and move the domain with the higher $\tilde{\tau}_{v,i}$ to the (at that time still) idle CPU (which after the transfer becomes the hot one).

We could also swap one VCPU from the hot CPU with the VCPU on the idle CPU, this is equivalent to staying well below $w_{ideal}$ ($w_{moved} = 0$ and $w_{ideal} = 0.5$) and it involves moving the VCPU with low $\tilde{\tau}_{v,i}$ to the idle cpu and the VCPU with high $\tilde{\tau}_{v,j}$ to the loaded CPU.

As one can see, the former case involves actions contrary to intuition, but works equally well. However as the general principle of the balancer is to move domains with low $\tilde{\tau}_{v,i}$ to an idle CPU, the latter approach is used. The natural question arising is: How does the scheduler decide which $\tau$ is "high" and which one is "low"? Of course an easy local measure is to simply compare (or better sort) taus on each CPU. But it is often more preferable to have a global standard to compare against. $\tau_{ideal}$ serves exactly this purpose and allows us to find the appropriate time, when to switch domains in the mentioned example.

**Selecting pairs of CPUs** As silently assumed, we limit the scope of the balancer to balancing between pairs of CPUs. This allows for the finding of $w_{ideal}$, which gives guidance on how much weight should be moved. The selection of pairs of CPUs becomes of course a crucial step in the balancer operation now. As getting close to $\tau_{cpu,k} = \tau_{cpu,l}$ is critical balancing process, it would be advisable, to balance between CPUs with the highest and lowest tau, in order to reduce imbalances. This is the main idea, further refinement to this will be made in the following. The available $\tau_{cpu,k}$ on each CPU $k$ is simply calculated using $t_{avail,k} = \Delta t - t_{rt,k}$, where $t_{rt,k}$ is maintained by the scheduler, which just adds

up time used by running real-time domains. To even out jitter in $\Delta t$, which can be introduced due to the balancer being scheduled not exactly at the same position during each period (see subsection 4.4.3), this value is projected on $\Delta t_{cpu-sample}$, and recursively averaged, so that we get:

$$t_{avail,proj,k} = t_{avail,k} \cdot \Delta t_{cpu-sample}/\Delta t$$

and further:

$$\tilde{t}'_{avail,k} = 1/2 \cdot \tilde{t}_{avail,k} + 1/2 \cdot t_{avail,proj,k}$$

$\tau_{cpu,k}$ is then simply determined by summing up the weights of VCPUs on cpu $k$ to get $w_{cpu,k}$ and then just $\tau_{cpu,k} = \tilde{t}_{avail,k}/w_{cpu,k}$. The set of $\{(k, \tau_{cpu,k}) \mid \text{for all CPUs } k\}$ of CPUs and their taus is then sorted by tau and balancing proceeds between CPUs, starting from both ends of the sorted set. If the difference between taus is below a threshold ($\Delta\tau_{thresh}$), no balancing between the selected CPUs occurs.

Note how this calculation takes varying time-consumption of the rt-VCPUs into account and thus allows for flexible placement of be-VCPUs.

**Finding swap-sets** Once we have found a pair of CPUs to balance, we can compute $w_{ideal}$ and then try to find a set of domains to move which has $w_{total} \approx w_{ideal}$. As mentioned, this is an instance of the knapsack (subset sum) problem, which is known to be NP-complete. Therefore the balancer approximates the solution by trying to find a close solution, by using multiple phases find an approximate solution.

In the first phase, all VCPUs are marked (put into the swap-set) according their relation to $\tau_{ideal}$. On the idle ($\tau_{cpu,k}$ is high) CPU, VCPUs are selected, which have $\tau_{v,i} > \tau_{ideal}$, on the hot ($\tau_{cpu,l}$ is low) CPU those with $\tau_{v,j} < \tau_{ideal}$, always ensuring that no violation of each VCPU's "cpumap" occurs. This is based on the assumption, that both CPUs continue to be hot or idle and do not swap their roles. This (as already established) relies on not exceeding $w_{ideal}$ by far. We check the total weight of the swap-set, and if it lies close to $w_{ideal}$, we simply carry out the migration of the domains.

If that is not the case, VCPUs are removed from the swap-set during the second phase, trying to bring $w_{total}$ closer to $w_{ideal}$. Domains get removed by increasing distance from $\tau_{ideal}$, *i.e.*, we start to remove those domains from the swap-set that received close to ideal service. Domains that were supposed to get moved from the hot to the idle CPU are kicked, if $w_{total} >> w_{ideal}$, those that should get moved in the opposite direction are removed, when $w_{total} << w_{ideal}$.

Please note, that *adding* VCPUs to the swap-set would not be helpful (although it could benefit $w_{total}$),

because we made sure in the first step, that all VCPUs that can benefit from a migration are already inside the swap-set. Trimming the swap-set continues, until we either find $w_{total}$ close enough to $w_{ideal}$, or the swap-set is empty. In the former case all marked VCPUs get moved to the other CPU and balancing continues with another pair of cpus. In the latter we start the third phase, which presses harder to actually achieve $w_{ideal}$. As you have noticed, neither the first phase nor the second are optimised for finding the proper weight. They rather try to find VCPUs which are most suitable to migrate first, migrating those, that have received different service than what the system can provide overall, estimated by $\tau_{ideal}$. In the third phase we actually try to get as close as possible to $w_{ideal}$, taus of the domains become a secondary concern, but are (of course) not ignored!

First of all, the swap-set is restored to the state before the second phase. In case $w_{total}$ exceed $w_{ideal}$, we first try to find one "heavy" domain on the hot CPU, that reduces (by being thrown out) $w_{total}$ to the interval of $[0, w_{ideal}]$, reasoning behind this is that it is very likely, that this VCPU's demands won't be satisfied on the other CPU either, as it just might be too heavy so that it actually would need two or more CPUs in parallel to get enough CPU-time.

Searching for domains by weight is done in (first-)best-fit, and returns the first domain, that is closest to a supplied parameter ($w_{close}$) and within the interval $[w_{close}, w_{thresh}]$ or $[w_{thresh}, w_{close}]$, where $w_{thresh}$ is also supplied by the caller. This means that $w_{close} = w_{total} - w_{ideal}$ and $w_{thresh} = w_{total}$ for this initial search-problem. When a VCPU is found, its removal from the set of VCPUs to be migrated will result in $w'_{total} \in [0, w_{ideal}]$, as requested above. If we cannot find such a VCPU, we continue with repeatedly removing lighter VCPUs that were supposed to get moved from hot to idle, by setting the search parameters to $w_{close} = w_{total} - w_{ideal}$ and $w_{thresh} = 0$, thereby finding VCPUs, that would bring $w_{total}$ closer to $w_{ideal}$.

VCPUs are removed from the swap-set and those with $\tau_{v,i}$ close to $\tau_{ideal}$ are removed first. This is achieved by walking the sorted (by $\tau_{v,i}$) list of domains in different directions, if try to find VCPUs that were supposed to be moved from hot to idle, the list is walked backwards (to smaller $\tau_{v,i}$), because if we have multiple best-fits, the first one is returned, and as it is the one with the highest $\tau_{v,i}$, thus it is also the one closest to $\tau_{ideal}$ (because all $\tau_{v,i} < \tau_{ideal}$). The inverse argument holds for the opposite direction.

Once we have achieved that $w_{total} \leq w_{ideal}$, we try to move it as close as possible towards $w_{ideal}$, by deleting domains from the idle-to-hot direction, just as much that $w_{total}$ will stay below $w_{ideal}$. If we finally get close to $w_{ideal}$, VCPU migration is carried out as usual. If we however end up with the empty set in one of the stages, nothing is balanced.

### 4.2.3 Special Features

Various of enhancements have been made to the core balancing algorithm, to further react to real-live behaviour and to provide better balancing characteristics.

**Awareness of blocking behaviour** Best-effort VCPUs that are either idle or have a highly I/O driven workload may end up not using their share of the available time on a CPU, because of blocking, more precisely $t_{recv,i} < w_{v,i} \cdot t_{avail,k})/w_{cpu,k}$ for $v_i \in bedoms_k$. This leads to an artificially low $\tau_{v,i}$ for these VCPUs, which (very likely) cannot be compensated for by moving the domain to another CPU. In order to account for this, the weight of the VCPU is scaled (for balancing purposes) linearly according to the amount of time the domain was blocked in an interval, *i.e.*, $\hat{w}_{v,i} = w_{v,i} \cdot (\Delta t - t_{blocked,i})/\Delta t$ and averaging is applied to this number too to adjust for more global blocking behaviour. Of course this can lead to domains with fractional weight and therefore fixed-point arithmetic is used when dealing with weights.

The scaled weight $\hat{w}_{v,i}$ now becomes the standard figure in the balancing algorithm above *e.g.* the total weight $\hat{w}_{cpu,k}$ on CPU $k$ is the sum of all the scaled weights of all best-effort domains on it: $\hat{w}_{cpu,k} = \sum_{v_i \in bedoms_{cpu,k}} \hat{w}_{v,i}$. In total this extension makes the balancer more flexible and able to deal better with I/O-based or idle domains, as it allows for better prediction of the impact a migration of an I/O-driven domain to another CPU.

**Cross-SMT performance penalties** Some modern CPUs provide SMTs [TEL95], performance enhancing measures, that allow for two simultaneous control flows to be executed on a single processor with a single set of most of the execution units, such as the ALU. Independence and diversity between the two flows can improve the utilisation of the core's functional units and therefore allow for higher total throughput. SMTs are usually presented to the system software layer as another core or even as another SMP. This is a reasonable abstraction, minimising the adaptation effort drastically. It is however, obvious, that these two pseudo-SMPs do not behave like two distinct real SMPs. Performance crosstalk has of course been an issue in standard SMP setups, be it contention for system resources, such as system-bus, memory and shared higher-level caches [BP04]. The expected performance impact for standard workloads is, however, limited to well-understood dimensions and does not affect most instructions. Examples for those performance penalties are: Locking, shared access to identical memory regions, contention for I/O devices. As those penalties are highly application specific, they are usually dealt with in the applications, too (lock-free synchronisation, data-decomposition, , *etc.*) receiving only little support by the underlying OS (the most notable exception being gang-scheduling [FJ97], to support

spinned locking).

In an SMT, the situation is much different, contention is taking place for pretty much all functional units in the processor. This has led to interesting ideas dealing with covert channels ([Per]). As this issue affects workloads of all kinds, SMP-schedulers of operating systems have to be made aware of the fact, that they might deal with SMTs, that behave differently from SMPs. Recent work [BP04] has taken a closer look on the performance penalties for applications running on SMTs compared to execution on a SMP. As a rough estimate a performance impact of about 30-40% was observed by the original author.

As the balancer relies on careful performance analysis for each CPU, it was necessary to account for these cross-SMT performance penalties. This is done by scaling times according to SMT-activity. Of course, the amount of time a VCPU spends on the CPU does not change, when another is making heavy use of the other SMT on the same CPU. But as time is the main accounting figure, it ought to be representative for the amount of work a VCPU can get done.

Scaling of times needs to be done at two places: Firstly when we account the amount of time a be-VCPU received during an interval, and secondly when we estimate how much time is available on a CPU for the execution of be-domains.

The general principle is the same for both: If we want to know for an interval $[t_1, t_2]$ how long concurrent execution on SMT-core $k_a$ took place, we take samples $(t_{id,1}, t_{id,2})$ of the CPU-time the idle-VCPU received on the *other* SMT-core $k_b$ of CPU $k$ at time points $t_1$ and $t_2$. As the idle domain just halts the SMT-core, it does not compete for (a significant amount of) the resources of the CPU.

Thus if $t_{id,2} - t_{id,1} = t_2 - t_1$, we can deduce that there was no concurrent SMT execution on core $k_a$. If $t_{id,2} - t_{id,1} = 0$, there was concurrent execution on the other SMT-core $k_b$ for the whole interval $[t1, t2]$ and so forth. In order to estimate the scaled time for a VCPU, we must therefore use:

$$\hat{t}_{recv,i} = t_{recv,alone,i} + \alpha \cdot t_{recv,conc,i}$$

Where

$$t_{recv,alone,i} = t_{id,2} - t_{id,1}$$

$$t_{recv,conc,i} = (t_2 - t_1) - (t_{id,2} - t_{id,1})$$

and $\alpha$ is the expected reduction factor due to concurrent SMT access, set to around 60-70%. The interval borders $t_1, t_2$ are the times, when VCPU $v_i$ was scheduled and descheduled, please note, that in case $t_{id,2} - t_{id,1} = t_2 - t_1$, we do actually get $\hat{t}_{recv,i} = t_{recv,alone,i} + \alpha \cdot t_{recv,conc,i} = t_2 - t_1 + \alpha \cdot 0 = t_{desched} - t_{sched}$, which is the original intention.

Once times have been modified in this way, the balancer can fully utilise the power of SMTs, by reacting to execution behaviour on one SMT of a CPU and then adapting the load on the other.

There is one little pitfall though: In the stage of the balancer, where the pairs of CPUs are selected, care has to be taken not to overload the CPU. This may happen, if we have a CPU with two idle SMTs. Naturally the balancer would decide to put VCPUs on both SMTs. However, fully loading one SMT decreases the performance for the other, but as the balancer cannot check that yet (there is no way to predict utilisation behaviour on the first SMT-core) it might overload the second SMT and the first one, too (because that one was filled, assuming no concurrent execution from its sibling). As a simple workaround the balancer does not balance to a SMT-core, if it has already balanced to the other SMT-core during this run of the balancer.

**CPU group detection and separation** Domains may use a "cpumap" bitmap, which restricts the CPUs, on which VCPUs of this domain can run. This bitmap therefore needs to be respected by the balancer. In order to make balancing possible, the balancer creates a map of all cpus a CPU can currently move domains to. This simply is $cpumap_{cpu,k} = \bigvee_{v_i \in bedoms_k} cpumap_{domain(v,i)}$, where $\bigvee$ is the bitwise or-operation of all specified elements and $cpumap_{domain(v,i)}$ the bitmap of the domain, VCPU $v_i$ belongs to. When the balancer has picked a pair of hot-idle CPUs, intended for balancing, it checks, whether the corresponding bit for the idle CPU is set in $cpumap_{cpu,hot}$. If set, there exists at least a single VCPU, that can be moved from the hot to the idle CPU.

We can also find separate groups of CPUs, between which no VCPUs can migrate. Those are characterised by

$$cpumap_{cpu,k} \vee cpumap_{group,x} = cpumap_{group,x} \quad \text{for } cpu_k \in group_x \tag{4.5}$$

and

$$cpumap_{cpu,k} \wedge cpumap_{group,x} = 0 \quad \text{for } cpu_k \notin group_x \tag{4.6}$$

Construction of them is straightforward, we incrementally create groups by iteratively checking for the above conditions and merging groups accordingly.

Each $cpu_k$ is checked with rule (4.6) to see whether it could fit into an existing group. If such a group is found, its "cpumap" might need an update because of (4.5), by simply using this rule to determine the updated "cpumap". If $cpu_k$ fits into two groups, these groups are merged, for example this might lead from figure 4.2 to figure 4.3, because of a domain with a VCPU on CPU 1 might also migrate to CPU 3. If it does not fit in any group, it creates a new group $group_z$ with $cpumap_{group,z} = cpumap_{cpu,k}$.

Once we have found those groups, the balancer just needs to deal with CPUs in each group and therefore

balancing complexity can be further reduced, but still care has to be taken when selecting pairs of CPUs, as for example in figure 4.2, CPU 0 and CPU 2 cannot exchange any of their VCPUs.

The groups also create a more realistic notion of $\tau_{ideal}$, as it is now computed for each of them separately, rather than for the whole system. It is therefore a more realistic measure and serves as a better guideline for finding distributions of domains.

The automated group detection also allows users to specify advanced balancing schemes, such as: keep all first SMT-cores of each CPU reserved for real-time domains (or domain zero) and restrict be-doms and their balancing to the second SMT-core of all CPUs, by simply setting the "cpumaps" of all be-domains accordingly. As cross-SMT penalty awareness is still active, the be-doms can react to changed SMT-performance caused by varying utilisation of guarantees by the rt-domains.

Another example are well known scenarios where groups of CPUs are dedicated to a single group of domains, isolating them from potential interference with other domains.

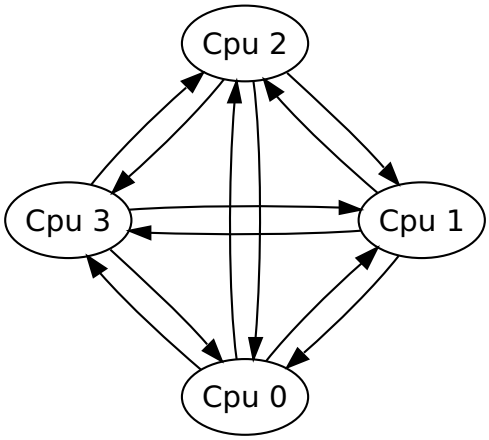


Figure 4.1: All CPUs in the system can balance to each other, thus they form a single balancing group.

**Mixed-mode domains** VCPUs that get guarantees from the scheduler (rt-VCPUs) can also apply for extra time, they are called *mixed mode* VCPUs, short mx-VCPUs (see subsection 3.2.4). These VCPUs cannot, however, be migrated by the balancer, because of the partly real-time execution, but the balancer has to take their presence on the CPU into account, by adding their (possibly implicitly specified) weight to the total weight present on the CPU they are running on. And of course, the execution time of the real-time execution part of these VCPUs is taken into account in $t_{rt,k}$ (which is needed to find $t_{avail,k}$ and $t_{avail,proj,k}$ respectively).

With this treatment (accounting, but not moving), $\tau_{cpu,k}$ for a CPU $k$, which contains mx-VCPUs is reduced, taking increased demand for extra-time by the mx-VCPUs into account.

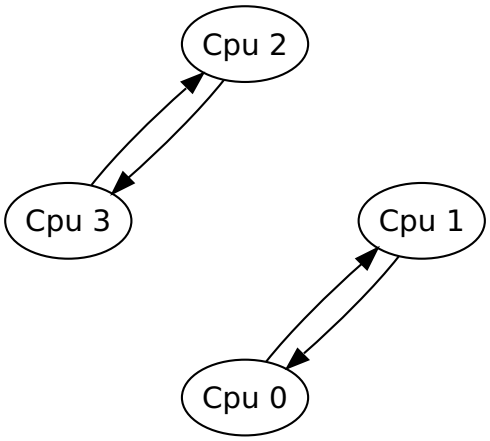


Figure 4.2: CPUs are separable into two balancing groups, as each VCPU of one balancing group can only be moved to a (physical) CPU belonging to the same group.

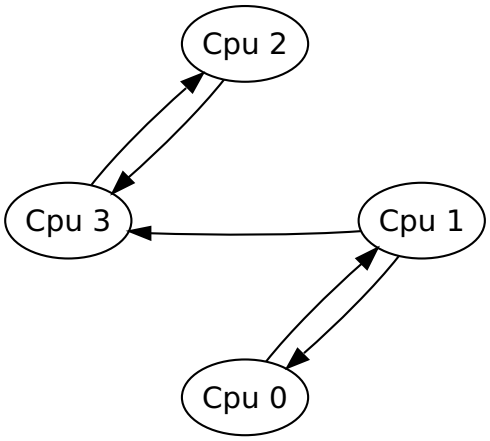


Figure 4.3: Some VCPUs on CPU 1 can also be moved to CPU 3, thus breaking the strict separation of the balancing groups. Hence all (physical) CPUs belong to the same balancing group. Although CPU 0 and CPU 2 now belong to the same balancing group, they cannot be chosen as hot-idle pair, as there are no VCPUs to move from one to the other.

## 4.3 Results

Final benchmarks of the balancer have not been created yet, as there is still some work to do, to clean up the code and adapt the balancer to the quickly changing code base of Xen. This turned out to be a difficult task, due to the strong dependence on mini-domains, which require some modification at key positions in Xens core. The features presented have not been thoroughly tested with real live workloads, but rather with very short micro-benchmarks, that just confirmed a correct operation. However, a single test-scenario was recorded, namely the test of the first working development version of the balancer. The test setup is very simple, three domains with a single VCPU each run in best-effort mode with weight 1, *i.e.*, they do not have any real-time execution part. Domain 0 is given a 5ms-every-20ms guarantee. Each of these four domains run the mentioned "slurp" tool, measuring CPU-time received by the domain. The four domains run on a 2.4 GHz Xeon server with two physical CPUs, without any additional SMT-cores.

As can be seen from the results in figures 4.4 and 4.5, the tested early version of the balancer already works as expected, for this simple workload. Figure 4.4 shows how the received CPU-time for each be-VCPU is fluctuating, depending on which physical CPU it is currently running on. Long-term service to each of the VCPUs is equal, as can be seen from figure 4.5, where the received CPU-time is accumulated over time.

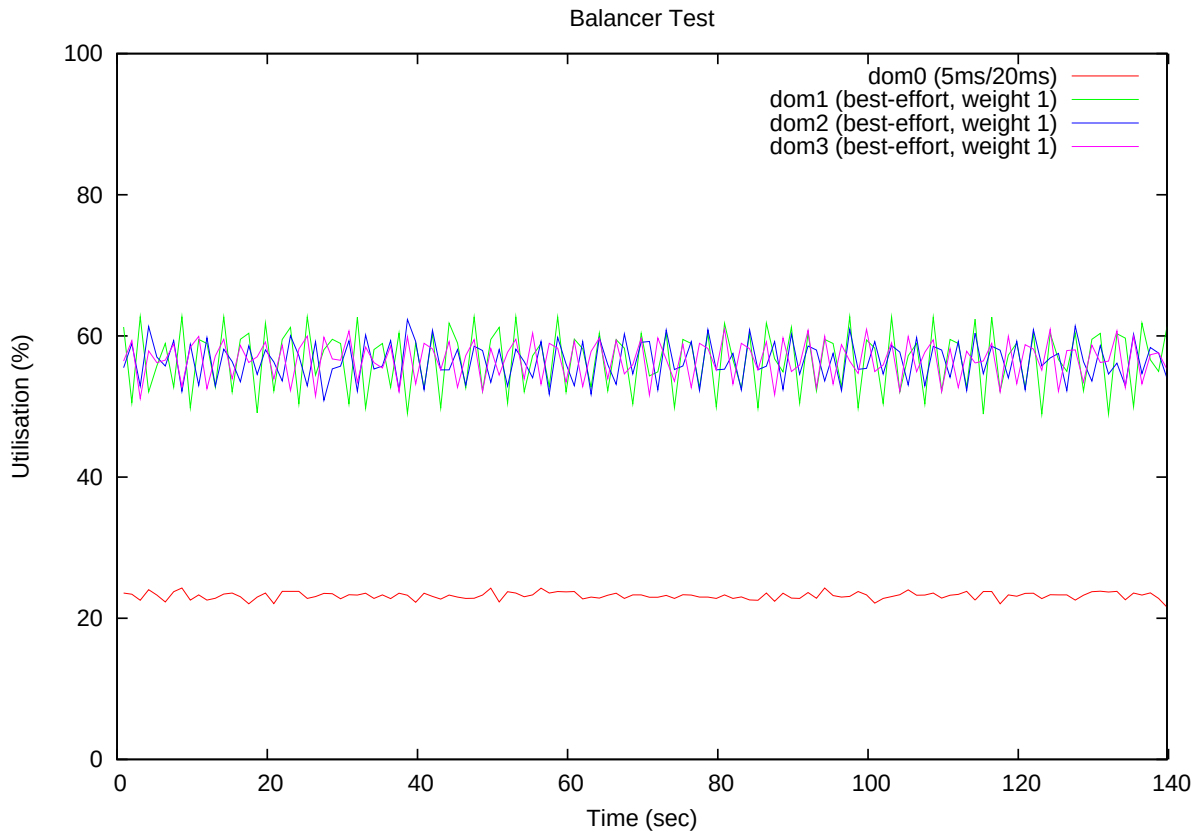


Figure 4.4: Balancer managing 3 best-effort VCPUs on a 2-way SMP-machine.

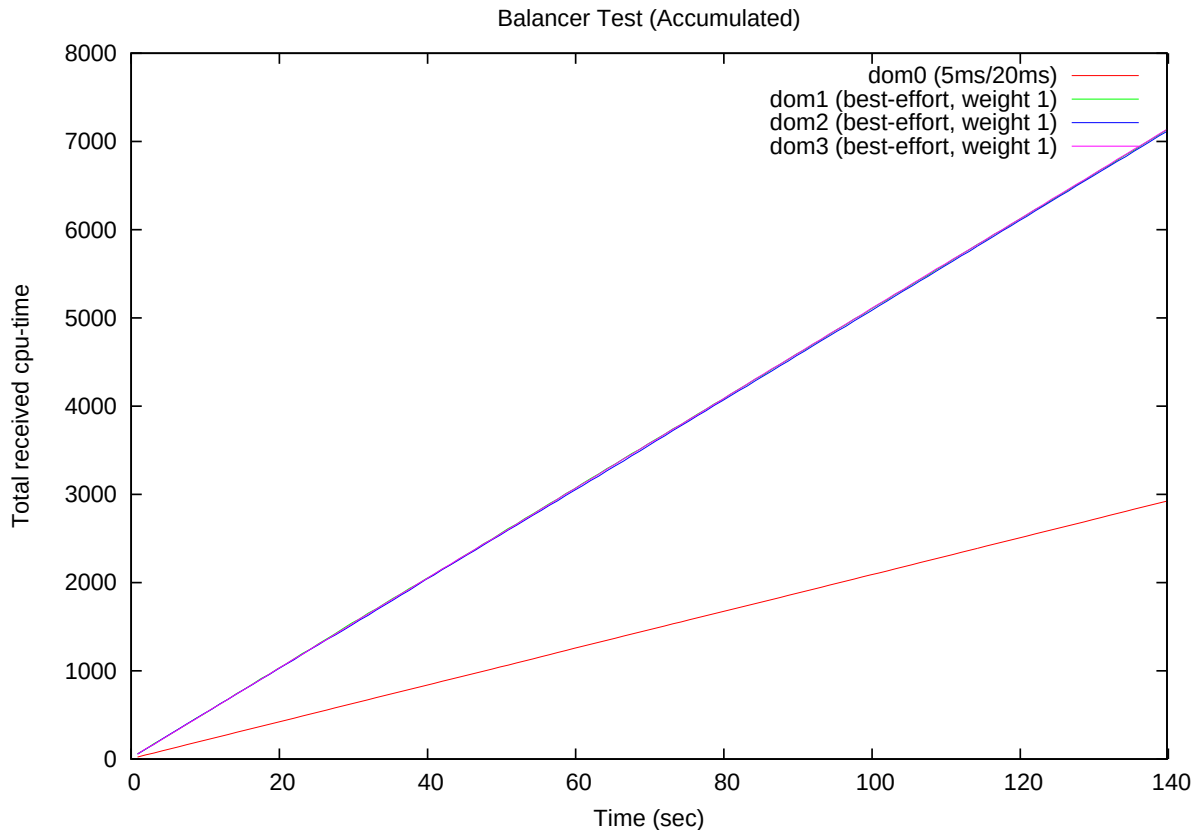


Figure 4.5: Balancer managing 3 best-effort VCPUs on a 2-way SMP-machine, showing accumulated CPU-time. The accumulated CPU-times for all three domUs overlap exactly, showing how they all receive identical service over time on different CPUs.

## 4.4 Infrastructure

### 4.4.1 Concept of mini-domains

In order to allow for an independent execution of the balancer, the concept of a *mini-domain* was introduced. A mini-domain is a domain that only executes code in ring 0, *i.e.*, Xen hypervisor code, therefore it does not own any kernel-/ user space state and memory. It can however be scheduled by Xen's scheduler, which can allow for timely execution (provided the scheduler has real-time capabilities, as the default SEDF scheduler does). Initially it was planned to keep the load balancer inside Xen's standard scheduling code, but this posed problems with locking, context migration and accounting, as the balancer would have been running in the context of the current domain. Migrating this domain to another CPU (which would have been standard balancing behaviour) by "itself" was impossible to fit into the existing code base without a complete rewrite of the locking and context switch code.

The current approach is based on the idle task, which has the aforementioned properties (no kernel-/user space state) already. Changes had to be made in the context switch code, in order to not check for the idle task in a hard coded way, but rather honour the "ring0-only" domain flag, which both, idle-task and mini-domains have set. To the scheduler a mini-domain is equal to any other "normal" domain in the system, *i.e.*, it can have deadlines, extra-time execution, the only distinction is made when the balancer mini-domain is initialised: Its timing parameters are set. See figure 4.6 for a schematic.

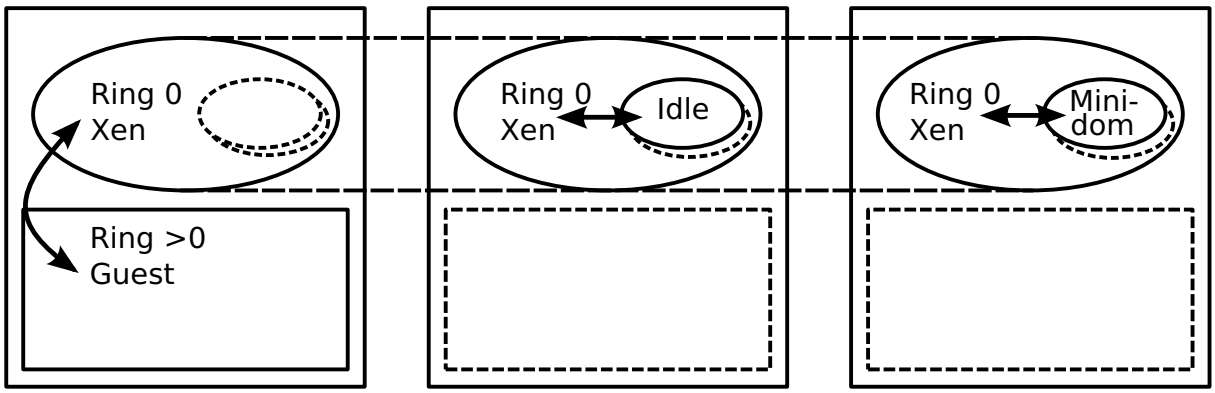


Figure 4.6: Comparing regular domains (left), the idle-domain (center) and a mini-domain (right). Xen's code is mapped into all three address-spaces, the idle- and mini-domain do not contain any guest-code.

### 4.4.2 Locking

As the balancer can now run almost anytime and parallel to the existing scheduling code, careful locking of data-structures is required. The balancer uses in total three locks that coordinate operation between itself and other scheduling components.

The *balancer-lock* guards the execution of the balancer itself and ensures that only one instance of the balancer is active at each time. It is therefore a single, system-wide entity.

Each CPU has a *best-effort-lock*, that serialises access to the *best-effort-queues*, which themselves are CPU-local. Additionally, there is a *schedule-lock* on each processor, that orders calls to most of the scheduler functions, including `do_schedule`, `wake` and `sleep`. The main balancing function is only protected by the balancer-lock, but once a pair of CPUs to balance between is selected, the balancer locks the best-effort-locks on both CPUs. It uses a very careful approach, *i.e.*, it tries to acquire both locks at once and does not hold the first lock and spins on the second, but rather releases the first lock, once the second lock is not available. This is necessary, as the locks are used in the scheduling code, which is timing-critical and holds the locks for a very short time only. As a natural effect, the balancer's code is *not* deadlock safe, because it acquires locks in arbitrary order; but this does not create a problem, as it is the only code that locks two best-effort-locks, and this code can only be executed on one CPU, because it is protected by the balancer-lock. Therefore the cyclic wait condition, which is necessary for forming of a deadlock, is void.

### 4.4.3 Scheduler support

The balancer is currently not independent from the scheduler, but rather relies on it to take samples of duration of execution and give an estimate on cross-SMT (see subsection 4.2.3) affection of CPU-time. Therefore the descheduling, wake and sleep code has been enhanced to gather these values, which are often already computed for the scheduler's purposes, but need to get stored again for the balancer, due to different intervals of summation and reset. Additionally, every $\Delta t_{VCPU-sample}$ (set to 100ms), the gathered accumulated timing data is averaged and stored as $t_{recv,i}$ for VCPU $v_i$ (see subsection 4.2.2 for details on how these values are used).

As already mentioned, the balancer relies on the real-time properties of the SEDF scheduler, as it gets executed every $\Delta t_{bal-period}$ which defaults to 50 ms, but the main balancing function is only called every $\Delta t_{bal-run}$ (500 ms). The tenfold oversampling guarantees low jitter in the actual times of balancer execution, as the real-time-scheduler guarantees that a domain will get its requested execution time in the period, but not when. Thus setting the period naively to 500 ms can result in an average jitter of

$500 - \epsilon$ ms (0 ms delay between two executions of the scheduler, when executed at the end of the first and beginning of the second period, and $1000 - \epsilon$ ms delay vice versa). This is a shortcoming in the task model of edf-based schedulers, which was proposed in [LL73]. Jobs with deadlines shorter than their periods (such as the balancer) are often treated as sporadic jobs. Treating those jobs with the normal earliest-deadline first algorithm produces satisfactory results, although the schedulable utilisation drops below 1. Our approach simulates standard edf behaviour for tasks with deadline $\neq$ end of period.

This tenfold oversampling is not only used to reduce jitter in the balancer's execution, but also serves the purpose of gathering timing information by the balancer. Similar to sampling timing data of VCPUs, the balancer requires timing data for the physical CPUs present in the system, it gathers these numbers every $\Delta t_{cpu-sample}$ (100 ms). Specifically, the time available for best-effort execution (extra-time) during the last sampling interval $t_{avail,k}$ on CPU $k$ is normalised and averaged and then used inside the balancer. The higher frequency of sampling of data values allows for a averaging process with higher precision.

## 4.5 Summary

### 4.5.1 Conclusion

This chapter has introduced the foundation for an experimental SMP-balancer for the Xen-hypervisor. This balancer supports guests running with best-effort guarantees and ensures, that each of these guests receives the same service according to a weight parameter, regardless of CPU utilisation by real-time guests and unforeseen behaviour by other guests.

The balancer achieves this goal by checking available slack time on each CPU and moving guests when they could receive better service on another CPU, additionally guests are "juggled" on the CPUs, if an ideal placement is not achievable, equaling long-term service.

Features of the balancer include support for cross-SMT performance penalties and adaption to blocking behaviour of guests. An automatic detection of balancing groups is also included, allowing simple implicit specification of well known partitioning schemes.

### 4.5.2 Outlook

The works on the core of the balancer have now pretty much finished. All of the mentioned features have been included and are in usable condition. However, there is still quite some work to do. First of all, more tests (other than domain start-up and "slurp") have to be evaluated to check balancer behaviour and the efficiency of the taken measures. I expect this to result in further refinement of given constants,

*etc.* Stability has improved a lot and the balancer's infrastructure is now assumed to be stable, however testing on more diverse configurations of machines and workloads is required to underpin this.

Unfortunately, the balancer has not yet been included to the Xen's mainline unstable repository, together with the fact that the author could not manage the code any further since August 2006, a merge of the repositories will be a demanding task, especially due to the newly introduced concept on mini-domains, which, though being rather straight-forward in nature, requires various small modifications at key code-lines in Xen (such as context switching).

Additionally, the code needs cleaning up and refactoring into smaller units, with a possible transition of the general balancing algorithm into user-land.

# List of Figures

# List of Tables

# Listings

# Bibliography

[AMD] AMD Processor Information - Virtualization. `http://www.amd.com/de-de/Processors/ProductInformation/0,,30_118_8796_14287,00.html`.

[Apa] Apache Webserver. `http://www.apache.org`.

[BDF+03] 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 Press.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference*, FREENIX Track, April 2005.

[BP04] James Bulpin and Ian Pratt. Multiprogramming performance of the pentium 4 with hyper-threading. *Third Annual Workshop on Duplicating, Deconstruction and Debunking (at ISCA'04)*, pages 53–62, June 2004.

[Cre81] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), September 1981.

[DC99] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 261–276, New York, NY, USA, 1999. ACM Press.

[FHN+04] Keir Fraser, Steve Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.

[FJ97] Dror G. Feitelson and Morris A. Jette. Improved utilization and responsiveness with gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer Verlag, 1997.

[GVC97] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.

[Int] Intel Virtualization. `http://www.intel.com/technology/virtualization/index.htm`.

[Int97] Intel Corporation. *Intel Architecture Software Developer's Manual - Volume 2: Instruction Set Reference*, 1997. Order Number 243191.

[Liu00a] Jane W.S. Liu. *Real-Time Systems*, chapter 3.3, pages 40–42. Prentice-Hall, 2000.

[Liu00b] Jane W.S. Liu. *Real-Time Systems*, chapter 6.8.2, pages 164–165. Prentice-Hall, 2000.

[Liu00c] Jane W.S. Liu. *Real-Time Systems*, chapter 4.6, pages 67–70. Prentice-Hall, 2000.

[Liu00d] Jane W.S. Liu. *Real-Time Systems*, chapter 6.3, pages 124–127. Prentice-Hall, 2000.

[Liu00e] Jane W.S. Liu. *Real-Time Systems*, chapter 6, page 115. Prentice-Hall, 2000.

[Liu00f] Jane W.S. Liu. *Real-Time Systems*, chapter 7.5, pages 233–243. Prentice-Hall, 2000.

[Liu00g] Jane W.S. Liu. *Real-Time Systems*, chapter 7, page 190. Prentice-Hall, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LMB+96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[MD78] A. Mok and L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. Seventh Texas Conf. Comput. Syst.*, November 1978.

[Mic] Microsoft VirtualPC. `http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx`.

[Per] Colin Percival. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`.

[RI00] J.S. Robin and C.E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[SWH05] Udo Steinberg, Jean Wolter, and Hermann Härtig. Fast component interaction for real-time systems. *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005.

[TEL95] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[ttc] TTCP - Network Testing Tool. `ftp://ftp.arl.mil/pub/ttcp/`.

[VMw] Vmware homepage. `http://www.vmware.com`.

[Web] M.A.M.E. - Multiple Arcade Machine Emulator. `http://www.mame.net`.

[WSG02] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, number 10, June 2002.

[Xen] Server Consolidation with Xen. `http://www.xensource.com/files/bundle_solution_brief.pdf`.

# A Raw measurements

## A.1 Raw Linux kernel compile results

| Domain 0 (20ms/20ms) | Domain 1 (1ms/20ms) | Domain 2 (2ms/20ms) | Domain 3 (5ms/20ms) |
|---|---|---|---|
| 739,00 | 12403,16 | 6156,68 | 2493,66 |
| 724,92 | 13350,19 | 6191,98 | 2520,34 |
| 731,28 | 12463,90 | 6170,20 | 2514,99 |
| 760,99 | 12435,11 | 6155,77 | 2496,31 |
| 729,45 | 12409,52 | 6159,36 | 2466,02 |

Table A.1: Execution times (in seconds) of non-concurrent compilations of a Linux kernel in different domains.

| Domain 1 (1ms/20ms) | Domain 2 (2ms/20ms) | Domain 3 (5ms/20ms) |
|---|---|---|
| 15273,87 | 6990,86 | 2732,38 |
| 15217,68 | 6964,83 | 2745,97 |
| 15159,52 | 6993,82 | 2728,38 |
| 15348,29 | 7004,30 | 2744,44 |
| 15233,74 | 6742,83 | 2728,76 |
| 15199,28 | 6952,09 | 2733,73 |
| | 6850,80 | 2747,57 |
| | 6942,86 | 2758,39 |
| | 6964,80 | 2717,17 |
| | 7027,60 | 2738,20 |
| | 6971,76 | 2765,61 |
| | 6972,97 | 2764,29 |
| | 6972,47 | 2748,15 |

| Domain 1 (1ms/20ms) | Domain 2 (2ms/20ms) | Domain 3 (5ms/20ms) |
|---|---|---|
| | 7080,05 | 2740,80 |
| | | 2780,99 |
| | | 2742,24 |
| | | 2743,20 |
| | | 2750,44 |
| | | 2738,12 |
| | | 2749,61 |
| | | 2764,66 |
| | | 2621,41 |
| | | 2739,38 |
| | | 2755,77 |
| | | 2713,98 |
| | | 2747,71 |
| | | 2767,52 |
| | | 2757,07 |
| | | 2607,09 |
| | | 2600,59 |
| | | 2605,35 |
| | | 2615,99 |
| | | 2715,26 |
| | | 2723,74 |
| | | 2744,01 |

Table A.2: Execution times (in seconds) of simultaneous compilations of a Linux kernel in different domains.

## A.2 Apache benchmark results

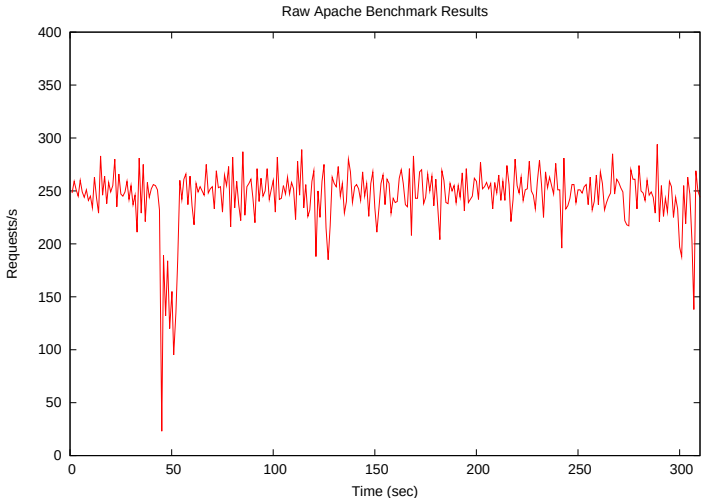


Figure A.1: Raw measurement data for Apache experiment 1, where Apache is running in domU and dom0's reservation is reduced every 30 seconds. See subsection 3.3.2 for details.

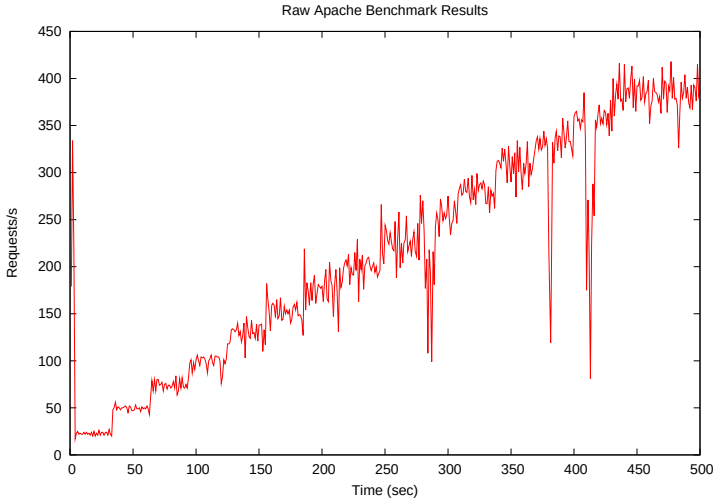


Figure A.2: Raw measurement results for Apache experiment 2, the reservation of Apache's domU is increased every 30 seconds and impact on throughput is analysed. See subsection 3.3.2 for details.

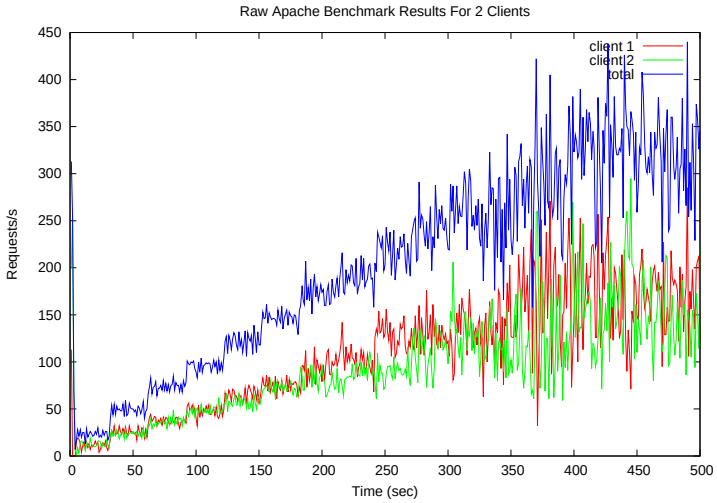


Figure A.3: Raw measurement results for Apache experiment 2, this time with two concurrent clients, in order to eliminate a possible client side bottleneck. See subsection 3.3.2 for details.

# B Source code

## B.1 Data structures

The following code-listings show the structures containing data for the key elements described in this document: Domains (B.1), VCPUs (B.2) and the common scheduler API (B.3). The first two are defined in `xen/include/xen/sched.h` and the latter in `xen/include/xen/sched-if.h`. Note the omission of architecture dependent parts for the VCPU and domain structures, these can be found (for x86 architecture) in `xen/include/asm-x86`. Additionally each scheduler can specify a set of private variable associated with each VCPU and physical CPU, these can be found in the respective c-files defining the schedulers (`xen/common/sched_sedf.c` for the scheduler discussed in this document and `xen/common/sched_bvt.c` for the included BVT scheduler, see subsection 3.2.2).

```
struct domain
{
        domid_t            domain_id;

        shared_info_t     *shared_info;     /* shared data area */

        spinlock_t         big_lock;

        spinlock_t         page_alloc_lock; /* protects all the following fields  */
        struct list_head   page_list;       /* linked list, of size tot_pages     */
        struct list_head   xenpage_list;    /* linked list, of size xenheap_pages */
        unsigned int       tot_pages;       /* number of pages currently possesed */
        unsigned int       max_pages;       /* maximum value for tot_pages        */
        unsigned int       next_io_page;    /* next io pfn to give to domain      */
        unsigned int       xenheap_pages;   /* # pages allocated from Xen heap    */

        /* Scheduling. */
        int                shutdown_code; /* code value from OS (if DOMF_shutdown) */
        void              *sched_priv;    /* scheduler-specific data */

        struct domain     *next_in_list;
        struct domain     *next_in_hashbucket;

        /* Event channel information. */
        struct evtchn     *evtchn[NR_EVTCHN_BUCKETS];
        spinlock_t         evtchn_lock;

        grant_table_t     *grant_table;

        /*
        * Interrupt to event-channel mappings. Updates should be protected by the
        * domain's event-channel spinlock. Read accesses can also synchronise on
        * the lock, but races do not usually matter.
        */
#define NR_PIRQS 256 /* Put this somewhere sane! */
        u16                pirq_to_evtchn[NR_PIRQS];
        u32                pirq_mask[NR_PIRQS/32];

        unsigned long      domain_flags;
        unsigned long      vm_assist;

        atomic_t           refcnt;

        struct vcpu       *vcpu[MAX_VIRT_CPUS];

        /* Bitmask of CPUs which are holding onto this domain's state. */
        cpumask_t          cpumask;

        struct arch_domain arch;

        void              *ssid; /* sHype security subject identifier */
};
```

Listing B.1: Structure containing management data for domains in Xen. From `xen/include/xen/sched.h`.

```
struct vcpu
{
        int              vcpu_id;

        int              processor;

        vcpu_info_t     *vcpu_info;

        struct domain   *domain;
        struct vcpu     *next_in_list;

        struct ac_timer  timer;         /* one-shot timer for timeout values */
        unsigned long    sleep_tick;    /* tick at which this vcpu started sleep */

        s_time_t         lastschd;      /* time this domain was last scheduled */
        s_time_t         lastdeschd;    /* time this domain was last descheduled */
        s_time_t         cpu-time;      /* total cpu-time received till now */
        s_time_t         wokenup;       /* time domain got woken up */
        void            *sched_priv;    /* scheduler-specific data */

        unsigned long    vcpu_flags;

        u16              virq_to_evtchn[NR_VIRQS];

        atomic_t         pausecnt;

        cpumap_t         cpumap;        /* which cpus this domain can run on */

        struct arch_vcpu arch;
};
```

Listing B.2: Structure containing management data for VCPUs in Xen. From `xen/include/xen/sched.h`.

```
struct scheduler {
        char        *name;              /* full name for this scheduler      */
        char        *opt_name;          /* option name for this scheduler    */
        unsigned int sched_id;          /* ID for this scheduler             */

        int          (*alloc_task)     (struct vcpu *);
        void         (*add_task)       (struct vcpu *);
        void         (*free_task)      (struct domain *);
        void         (*rem_task)       (struct vcpu *);
        void         (*sleep)          (struct vcpu *);
        void         (*wake)           (struct vcpu *);
        struct task_slice
                     (*do_schedule)    (s_time_t);
        int          (*control)        (struct sched_ctl_cmd *);
        int          (*adjdom)         (struct domain *, struct sched_adjdom_cmd *);
        void         (*dump_settings)  (void);
        void         (*dump_cpu_state) (int);
        void         (*notify_pincpu)  (struct vcpu *, int, int);
};
```

Listing B.3: Structure containing the definition of the common scheduler API in Xen. From `xen/include/xen/sched-if.h`.