Universität Karlsruhe (TH)
Institut für
Betriebs-und Dialogsysteme
Lehrstuhl für Systemarchitektur

Energy-Aware Scheduling of Virtual Machines in a Multiprocessor Environment

Marcus Reinhardt

Diplomarbeit

Verantwortlicher Betreuer: Prof.Dr. Frank Bellosa

Betreuender Mitarbeiter: Dipl.-Inform. Jan Stöß

13.Juni 2006

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 13. Juni 2006 Marcus Reinhardt

Abstract

This thesis presents a new concept to schedule virtual machines in a multiprocessor environment due to their energy consumption. We propose to divide the process of energy-aware scheduling in a virtual machine environment into two parts: guest-level scheduling and host-level scheduling. Thereby an energy-aware scheduler can benefit from the varying scheduling information available on both scheduling levels.

At first we measure the energy consumption of threads and virtual machines. We suggest to use the energy consumption of threads and virtual machines in combination with energy budgets to come to scheduling decisions. Furthermore we elaborated the framework required to implement energy-aware scheduling policies on both scheduling levels. High scheduling accuracy because of fine-grained energy profiles for threads characterizes guest-level scheduling. Host-level scheduling is equipped with mechanisms to overcome non-energy-awareness or malfunctioning of guest operating systems. Additionally it supports a better exploration of energy capacities of all CPUs in a multiprocessor environment by migrating complete virtual machines.

The flexible design of our framework supports the implementation of various energy-related scheduling policies, for example energy-aware balancing. Due to the two-leveled approach we can participate from fine-grained scheduling decisions on guest-level based on the energy consumption of threads as well as from scheduling decisions based on informations only available on host-level (for example the energy consumption of physical CPUs).

Acknowledgments

Zunächst möchte ich mich bei meinem Betreuer, Herrn Jan Stöß, für seine umfangreiche und fachlich herausragende Betreuung bedanken. Mein weiterer Dank gebührt Herrn Prof.Dr. Frank Bellosa, der mir die Gelegenheit zu wissenschaftlichem Arbeiten in einem interessanten und anspruchsvollen Umfeld gab.

Desweiteren möchte ich mich bei Herrn Christian Lang, Herrn Sebastian Biemüller und Herrn Dominik Dinius bedanken, die mir auf verschiedenste Weise bei der Anfertigung der vorliegenden Arbeit geholfen haben, sowohl durch Probelesen und fachliche Diskussionen als auch mit Gesprächen zu Themen eher philosophischer Natur.

Besonderer Dank gilt meinen Eltern, Renate und Heinz Reinhardt, meiner Schwester, Sylvia Reinhardt, und meiner Großmutter, Hilde Reinhardt, für deren stetige Unterstützung über die letzten Jahre, fortwährende Ermutigung in schwierigeren Zeiten und dem Probelesen aller von mir angefertigten Arbeiten.

Abschließend möchte ich mich bei meiner langjährigen Freundin, Johanna Müller, für ihre Lebensfreude, Treue und Zuneigung bedanken.

Contents

Chapter 1

Introduction

Virtualization as an approach for server consolidation is a promising idea to reuse and to better exhaust available hardware resources, especially as server hardware is very expensive. However, server hardware typically is not only expensive. It also consumes a lot of energy and thus requires considerable expenses for cooling the system. This further increases the costs for server. An example for this issue which recently gained public attention for cooling and energy costs was when Google's chief engineer reported the strong increasing of energy costs on Google's server farms [3]. Thus, if the virtualization technology shall be an alternative on the server market of the future, a framework controlling the energy consumption of virtual machine environments is indispensable.

Classic approaches to control energy consumption, for example frequency scaling, can not be employed in a virtualization environment: Different virtual machines eventually require different operating frequencies. An efficient mechanism which allows to agree on one frequency is not available. Approaches, which move the energy control mechanism from hardware level to software level, like the approach introduced in [12], which shows how to balance the power consumption of multiprocessor systems, are far more promising for virtual machine environments. We believe that especially the energy-aware scheduling demonstrated in [12], which utilizes event-based energy estimation introduced in [4, 5], is an approach with the potential to offer ideas to control energy consumption in virtual machine environments with additional support of multiprocessor systems.

1.1 Problem

Energy-aware scheduling is a complex process. Policies which schedule energy-aware require non-standard operations of an operating system. For example energy-aware scheduling leverages resource containers in order to account energy [1, 12]. Such a scheduler furthermore requires a mechanism to throttle energy consumption and to balance energy consumption.

The approach in [16] discusses basic issues concerning energy-aware scheduling of virtual machines. It offers a couple of solutions to these issues, for example energy accounting in virtual machines or direct reaction to a CPU's temperature. However, this paper is focused on throttling the energy consumption of threads within a virtual machine due to their energy consumption. Besides this paper no further work is known discussing energy consumption of virtual machines and consequently none exploring issues of multiprocessor systems in this context, too.

1.2 Approach

The non-disposability of a solution motivated us to explore a new framework required for energy-aware scheduling in a virtual machine environment with support for multi-processor systems. We propose to divide the issue into two subproblems. Firstly we suggest extensions to guest-level scheduling to control the virtual machine internal energy-aware scheduling. Secondly we propose new mechanisms for host-level scheduling to implement global energy-aware scheduling policies.

The separation between guest-level scheduling and host-level scheduling is based on the observation that the granularity of information available within a virtual machine environment differs significantly between guest and host. Fine-granular informations for individual threads are available only within an operating system's kernel. To exhaust these informations we introduce a number of new facilities we subsume to the term guest-level scheduling, for example energy budgets, which finally offer a fine-granular energy-aware scheduling within a virtual machine. Host-level scheduling is residing in the major component of a virtual machine environment: The virtual machine monitor. The virtual machine monitor has no access to the fine-granular thread information. But it has access to globally available, energy related informations like the CPU temperature or the battery state. Furthermore, it knows about the existence of all virtual machines; an advantage essential to an energy-aware scheduler which tries to even out the energy consumption of virtual machines or CPUs. Our approach equips the virtual machine monitor with a number of new operations sufficient to apply global information to the energy-aware scheduling process.

In order to evaluate our approach we implemented a prototype which demonstrates the practicability of our extensions to guest-and host-level scheduling. We have applied our design to the *L4Ka Virtualization Environment* developed by the System Architecture Group of the University of Karlsruhe [21]. We successfully extended the existing virtualization environment with our design ideas and extracted relevant evaluation data. On an Intel Pentium D clocked at 3 GHz we monitored the functionality of guest-level scheduling and host-level scheduling by analyzing different scenarios, for example energy balancing of physical CPUs using host-level scheduling. We furthermore proved that our design induces only a negligible energy overhead if implemented in a suitable way. Additionally we evaluated the performance of our major components to get an idea about the capabilities and about limits of these components. Finally we conducted experiments analyzing the network throughput in order to further demonstrate the different advantages of guest-level scheduling and host-level scheduling.

1.3 Outline

This thesis is structured as follows:

- **Chapter 1: Introduction**
  The introduction offers some basic reasoning for this thesis and gives an abstract view to our approach.

- **Chapter 2: Background**
  The background chapter reviews some major contributions to our approach in detail. We show the similarities and differences. Furthermore we partially reason some of our basic functionality.

- **Chapter 3: Design**
  Chapter 3 presents the design of our approach. We focus on a clear structure beginning with a coarse description of our solution and moving more and more into details with each section.

- **Chapter 4: Implementation**
  The implementation chapter offers a discussion of the implemented prototype. We focus on the major topics as a complete discussion is beyond the scope of this thesis.

- **Chapter 5: Evaluation**
  This chapter contains the evaluation data we produced with our prototype. It demonstrates the functionality in detail and for different situations. Furthermore it proves the low energy overhead created by our design. Finally we address the performance of our host-level components and discuss the advantages of guest-level scheduling and host-level scheduling.

- **Chapter 6: Conclusion**
  Finally we give a conclusion and offer some approaches for future work.

Chapter 2

Background

This chapter presents some background material closely related to our work. In some words we discuss virtual machines before we proceed to event-based energy estimation and the problems of energy estimation in virtual machines. Furthermore we take a look on migration as it is an important key mechanism for this thesis. We shortly outline Linux migration as well as live migration of virtual machines, which gave us some initial ideas in this context, especially for the implementation of our prototype.

2.1 Virtual Machines

The term *virtual machine environment* describes an architecture consisting of a *virtual machine monitor* and multiple guest operating systems being executed in parallel, yet completely isolated from each other in so called *virtual machines*. Code execution is separated into native instruction execution and virtual instruction execution. The instructions to be executed virtualized form the *sensitive instruction set* [15]. Virtualized instruction execution is to take a sensitive instruction and to replace it by code of execution adopted to the virtual machine monitor. This process is called code emulation. Sensitive instructions normally are related to resource management because resources (for example memory management or interrupt assignment) are shared by the guest operating systems. For example, it is quite comprehensible that two guest operating systems should not access the same part of memory. Virtualization environments require prohibiting them from accessing resources directly to achieve the guest isolation. Therefore sensitive instructions are replaced by a new set of instructions which inform the virtual machine monitor to execute the emulation code so that other virtual machines take no notice about it.

Approaches

Different approaches of virtualization have been elaborated. A technology called *pure-virtualization* is an approach which provides *emulation code* for each sensitive instruction. Non-sensitive instructions are executed further on directly by the hardware. The injection of emulation code allows to run unmodified guest operating systems. Unfortunately, this process leads to a significant performance loss as the virtual machine monitor needs to detect the sensitive instructions and afterwards has to replace them by emulation code. A virtual machine using pure-virtualization which recently gained a public audience is VMware Workstation [24].

The second approach called *para-virtualization* requires no further injection of emulation code for each instruction in the sensitive set. The virtual machine monitor provides a number of interfaces to the emulation code which has to be invoked by the guest for themselves. Thus detecting instructions which require the injection of emulation code vanishes. Thereby the performance of the code execution is increased. Additionally no switches between address spaces for code emulation are required, as parts of the virtual machine monitor are located in the address space of the guest operating system. However, the fact, that no further detection of sensitive instructions is utilized, implicates, that each guest operating system which shall run upon this extended architecture needs to be modified. The developer has to replace all sensitive instructions manually. This is a time intensive procedure which requires outstanding skills in operating system development. Para-virtualization is adopted by the L4Ka Virtualization Environment [21] and Xen [27].

A third approach is called *pre-virtualization*. This relatively new concept tries to reduce the engineering effort to port a guest operating system to a virtualization architecture which follows the para-virtualization approach. At the same time the authors try to solve the issue of performance loss which appears in pure-virtualized architectures. In [8] the authors demonstrate the reduction of engineering time required to port a guest operating system to a virtualization architecture. They developed a new pre-compiler which injects code fragments when the operating system is built. These code fragments simplify the detection of sensitive instructions and consequently they reduce the performance loss, too. A further considerable advantage of pre-virtualized operating systems is that they can be executed directly on hardware as well as on pure-virtualized architectures.

Conclusion

We implemented our prototype for a para-virtualized architecture. It induces the minimum performance loss to a system. Thus the loss of energy capacities due to the virtualization architecture can be expected to be minimal, too. Event-based energy estimation which we use for measuring energy [see Section 2.2.1] uses besides other parameters the performance of a system in the estimation process by considering the CPU's time stamp counter. Therefore in order to keep down the energy loss as low as possible using a para-virtualized architecture suggests itself.

2.2 Energy-Aware Scheduling

In this section we review *energy-aware scheduling*, its related components and scheduling algorithms which can be used with energy-aware scheduling in different manner. We begin with a summary of event-based energy estimation itself and continue by reviewing energy-aware scheduling of virtual machines. Afterwards we discuss some potential scheduling algorithms. Finally we elaborate the basic functionality required to realize such algorithms in a multiprocessor driven virtual machine environment.

2.2.1 Event-Based Energy Estimation

Event-based energy estimation is the state-of-the-art approach to measure energy consumption, first published in [5]. It offers a mechanism to determine an on-the-fly energy value from the CPU without the requirement of any further hardware.

The idea is leveraging the so called performance counters available on a processor counting energy related performance events. On a Pentium 4 CPU there are 48 different events and 18 counters available [18]. This partially reasons the success of this method as demonstrated in [5]. The basic proceeding is the periodical inspection of the deviance of 8 energy related performance events (for example mispredicted branches or unhalted cycles). Afterwards the energy consumption for an inspected period can be calculated. The deviance of each performance counter therefore is multiplied with a weight which is elaborated for the used processor type. An adequate adoption of this approach can provide the consumption of each thread running in an operating system [5].

2.2.2 Energy-Aware Scheduling of Virtual Machines

Energy-aware scheduling uses the energy consumption of threads to set up a schedule originally designed for SMP systems [14]. Threads are assigned to processors in order to balance the energy consumption of all system CPUs.

Energy-aware scheduling of virtual machines considers the new problems of the process addressed above for single processor machines running a virtual machine. Measuring the energy consumption of threads is no more reduced to the measurement of the energy consumption on a processor. The virtual machine technology permits the operation of multiple guest operating systems which may run multiple threads each. However, event-based energy estimation is still capable to offer a reliable approach to measure the energy consumption in a virtualization environment. Therefore the energy consuming source has to be identified on thread level and on virtual machine level. As we will demonstrate this implicates support of energy estimation in the virtual machine monitor. Classic energy estimation within each guest operating system would result into an accounting of the consumed energy to multiple guest operating systems [16]. Thus we have to correct the assignment of energy to threads and to virtual machines. In the design section we discuss an approach how to overcome this problem we call *virtualized performance counters*. We additionally discuss an implementation of this solution in Chapter 4.

2.2.3 Scheduling Algorithms

The implementation of scheduling algorithms executing an energy-aware scheduling of virtual machines in a multiprocessor environment requires a minimal set of mechanisms, which we will consolidate in Chapter 3. In this section we deduce this minimal set of functionality by reviewing some scheduling algorithms which can be reused for energy-aware scheduling of virtual machines in a multiprocessor environment in different ways.

Lottery Scheduling

Lottery scheduling is a randomized resource allocation mechanism. The utilization of a resource is represented by a ticket which is allocated to clients in a lottery [26]. Tickets are transferred to the client which won a lottery explicitly. The tickets are expressed by using a currency fitting to the resource the scheduler is controlling.

Lottery scheduling requires two features of a resource allocator. At first it has to offer a mechanism assigning a budget, as the ticket owner only may consume a partial amount of the resource. Second it requires a representation for this amount, the currency.

Imprecise Computations Scheduling

The class of the quality assuring schedulers contain a lot of schedulers required in modern systems, for example multimedia scheduler or real-time scheduler. We selected *imprecise computations scheduling* as it is the essential scheduling policies of this kind. This scheduling approach separates the resource's total quantum in two parts: The mandatory part and the optional part. The mandatory part has to be executed definitely within the considered period. The optional part may be executed if resources still are left following a predefined distribution function [11].

Imprecise Computation Scheduling requires two features of a resource allocator. At first it has to assign a mandatory budget which is available definitely in a period. At second it needs to assign an optional budget which can be exhausted after its mandatory one is consumed already.

Temperature-Aware Scheduling

In [14] *temperature-aware scheduling* has been introduced as an approach to balance the temperature of CPUs in a SMP system. It calculates the energy consumption for each CPU as addressed above. In the following the algorithm determines energy limits from predefined temperature limits. Afterwards it migrates threads from hot CPUs (CPUs exceeding their energy limit) to others. By repeating this procedure periodically the temperature of all CPUs can be kept balanced.

This scheduling policy requires a special feature of a resource allocator: The migration.

Conclusion

We have shortly reviewed 3 representative scheduling policies, which can be applied to schedulers in a virtual machine environment with multiple processors. We tried to deduce a minimal set of functions required for this purpose. Lottery scheduling requires the assignment of budgets and the availability of a representation of the assigned amount. Our design will adopt this by introducing energy budgets, which use energy as representative. Imprecise Computation Scheduling requires assigning of two classes of resource quantities: A mandatory part and an optional one. We will not go into details by offering two separated classes of energy budgets. In order to still address overload situations, we will introduce a recalibration mechanism. It allows to redistribute energy budgets from virtual machines with a lower energy consumption to virtual machines which have exhausted their energy capacities. Furthermore this policy needs to rely on the keeping of these budgets. Consequently we will offer a suitable mechanism adopted to our problem called suspension. Temperature-aware scheduling finally requires migration. Migration will be available within our design, if energy-aware scheduling can be applied. Recapitulating our design has to offer the functionality of energy budgets, suspension and migration if common scheduling policies shall still be applicable upon our design.

2.3 Migration

Migration is a basic concept we are strongly relying on in our design. We give a short overview on the related concepts. At first we review the Linux scheduler which supports the migration we reuse. Then we take a look at live migration of virtual machines, which however is only partially of interest for our problem. It contained some initial ideas when we implemented virtual machine migration for our prototype.

2.3.1 Linux Migration

The Linux scheduler offers migration in order to perform load balancing on the CPUs of a multiprocessor system. For each CPU the scheduler maintains a runqueue. This queue contains all threads on its related CPU currently runnable. Furthermore it keeps a variable which contains the current CPU load. The CPU load is calculated in dependency to the number of threads on this CPU. After a rebalance period has passed the scheduler inspects the load of the current CPU. It compares this value to the CPU load on all other CPUs. If there are significant differences (25% or more) between two CPUs the scheduler pulls over 1 or more tasks in case there are more than 1 task running on the source CPU.

Basically the Linux migration offers facilities we reuse later when we implement our prototype. But we have to modify the scheduler such a way it controls the CPU's energy consumption, not the CPU load. Additionally we will see in Chapter 4 that there are issues concerning the fluctuation of a CPU's energy consumption which require further modifications to the Linux scheduler.

2.3.2 Live Migration of Virtual Machines

Live migration of virtual machines offers a mechanism to move complete virtual machines including their operating system and their applications between different host machines. The design considers different phases to also migrate memory (the push phase, stop-and-copy-phase and the pull phase). The authors state, that implementations eventually would only rely on one or two of the described phases (exemplified by pure stop-and-copy migration, pure demand-migration and pre-copy migration). They additionally mention the problem of starving network services in case memory images are sent in a round-trip manner. The network bandwidth can be exhausted then. They consider the migration of local resources especially for network resources (open connections shall be kept without any forwarding mechanism). The design offers an six step protocol for the migration of virtual machines. It focuses on the required processes in order to completely migrate the virtual machine within a short period of time.

Live migration of virtual machines offers some ideas for our migration of local virtual machines we will discuss in Chapter 3 and 4. We will not consider issues arising from keeping open network connections available, migrating memory or the consequences of bandwidth loss. We will focus on virtual machine migration between different (local) CPUs, which yet includes some of the introduced migration stages (for example 0: pre-migration, 3: stop-and-copy or 5: activation).

Chapter 3

Design

In this chapter we introduce our solution for *energy-aware scheduling of virtual machines in a multiprocessor environment*. We start with the definition of our design goals followed by a short overview on our scheme. After this we consider the major parts of our design in more detail. Thereby we proceed in a way that we refine our ideas from section to subsection. In the last section we give a short summary including the verification of our design goals.

3.1 Design Goals

We begin by defining our major design goals our design has to reflect. We verify them within the scope of the summary in the last section of this chapter.

- **Accuracy.** As we are dealing with accounting and distribution of energy we need accuracy as a virtual machine must not be billed for more energy than it really consumed. Furthermore it has to be able to consume all the energy it has been assigned. Any malpractice leads to unjustifiable punishment, for example a virtual machine is suspended by the virtual machine scheduler, a thread is descheduled by the guest's scheduler.

- **Safety.** The energy balance of our system has to be kept even. Malicious virtual machines (for example virtual machines which are not familiar to energy-aware scheduling) must not be able to undermine our design's mechanisms. Consequently our design has to control correctly uncooperative virtual machines.

- **Flexibility.** We support the implementation of different scheduling algorithms (see Section 2.2.3) upon our design. Furthermore we allow scheduling algorithms to be located as well in the virtual machine itself as in the virtual machine scheduler. Thus, the advantages of the different scheduling levels (for example obtaining precise scheduling decisions or leveraging global scheduling parameters like the CPU temperature), which we introduce below, can be employed.

- **Efficiency.** Some of our design's components are executed very often within the guest kernel or within the virtual machine monitor in order to support the adequate accuracy for accounting and scheduling decisions. Therefore these components need to be designed as efficient as possible. Our design's components must consume only a little amount of energy. Furthermore it must not slow down the complete system significantly.

3.2 Our Scheme

This thesis introduces a novel approach for energy-aware scheduling of virtual machines within multiprocessor environments. We provide a framework which offers the mechanisms to schedule threads on virtual CPUs for the guest's scheduler and which additionally provides the support to schedule virtual machines to physical CPUs by the virtual machine scheduler in an energy-aware manner as depicted in Figure 3.1. The mechanisms are designed to be used for the implementation of different scheduling policies within both schedulers. In the following we call threads the *scheduling principals* and virtual CPUs *the scheduling resource* of the guest's scheduler. Virtual machines are considered as the *scheduling principal* of the virtual machine scheduler and physical CPUs as its *scheduling resource*. A further term is the *scheduling entity* which embraces the scheduling principal and the scheduling resource. We only refer to this term if the relation between scheduler, principal and resource is evident.

Our proceeding to consider both levels of a virtual machine environment, the guest-level and the host-level, offers the optimal exploration of the properties of the considered scheduling resources, the virtual CPUs on guest-level and physical CPUs on host-level. Scheduling policies can benefit from the informations available on their level. For example policies implemented in the guest-level scheduler can rely on fine-granular informations available for threads, for example the energy consumption of threads. However, a scheduling policy on host-level can consider informations which are only available globally, for example the temperature of physical CPUs. In order to transfer these benefits to our scheme we divide the design in two major parts: Guest-level scheduling and host-level scheduling.

The term *guest-level scheduling* generally relates to the scheduling process of threads within the virtual machine. We extend this term by introducing energy as the decision criterion for the guest-level scheduler. The guest-level scheduler is considered to be the manager of the guest's threads. It assigns them to its available virtual CPUs in an energy-aware manner. We present all required facilities to implement energy-aware scheduling policies already within a virtual machine. This covers energy budgets as well as a suspension mechanism and a mechanism for migration.

The term *host-level scheduling* describes the functionality for the virtual machine monitor to actively interfere to the energy consumption of virtual machines. Host-level scheduling manages the assignment of virtual machines to physical

CPUs and regulates the energy consumption of the virtual machines. We introduce the mechanisms which are necessary for the virtual machine monitor to implement energy-aware scheduling policies. Especially we consider the required key functionality called virtual machine migration and virtual machine suspension. These mechanisms allow to utilize energy capacities of physical CPUs on the one hand and enforce energy limits for physical CPUs on the other hand. We gain the possibility to react on free energy capacities by migrating a virtual machine and we have mechanisms to keep energy limits. In the following we discuss the concepts and ideas to be met by our design in detail:

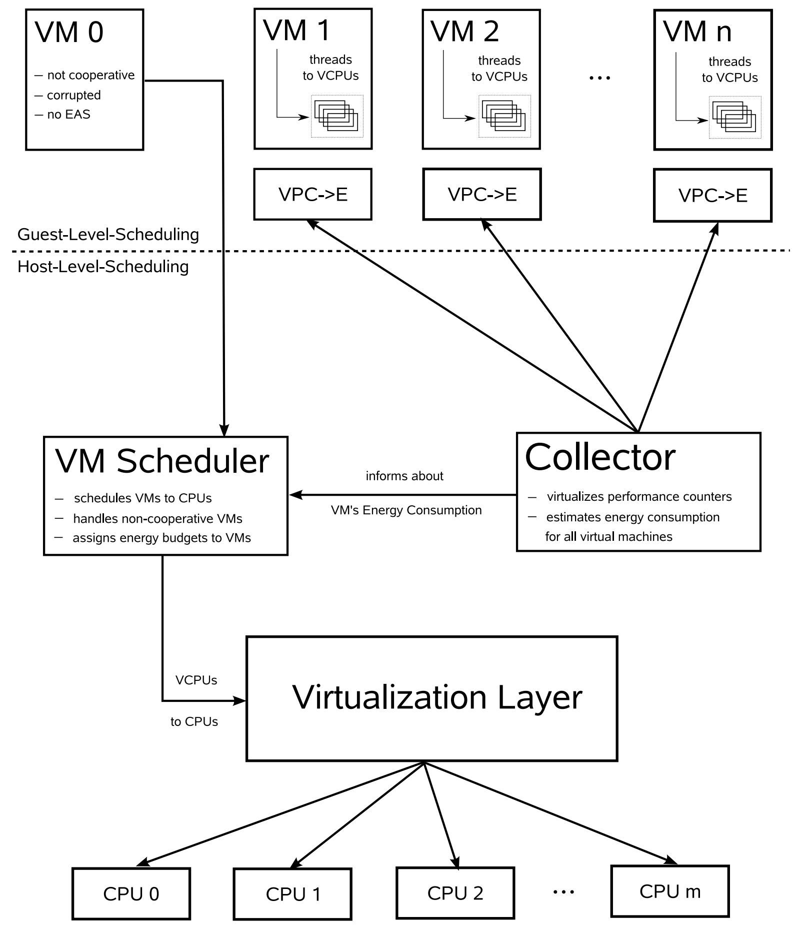

Figure 3.1: Abstract view to our design: The collector provides the virtualized performance counters for the consecutive energy estimation. The virtual machine scheduler may migrate the virtual machines between physical CPUs depending on energy budgets. The virtual machines may migrate threads to virtual CPUs due to the virtual CPUs' energy budgets. Virtual machines which are not scheduling energy-aware are kept within their boundaries by the virtual machine scheduler.

Guest-Level Scheduling. The virtual machine runs a guest operating system. Thus the guest operating system's scheduler maintains a fine-grained view to the properties of its threads and its virtual CPUs. Especially it knows about the energy consumption of its threads and its virtual CPUs. We use the energy consumption of threads and energy consumption of virtual CPUs as decision criterion for guest-level scheduling. Therefore, guest-level scheduling requires the energy consumption of threads and virtual CPUs, its scheduling principal and its scheduling resource. The energy consumption is derived from *virtualized performance counters*. They are supplied by a new component, called the *collector* which resides within the virtual machine monitor. It provides the virtualized performance counters as well as the energy consumption required to compile the energy profiles for host-level entities. Furthermore we introduce *energy budgets* which support the limitation of the energy consumption of a virtual machine. In order to solve overload situations we supply guest-scheduling with a *recalibration* mechanism allowing redistribution of energy capacities between the virtual machines.

Host-Level Scheduling. Guest-level scheduling is a feature located in the virtual machines themselves. However, some issues can not be solved by using guest-level scheduling. If a virtual machine fails to perform energy-aware scheduling, for example because they are not capable of energy-aware scheduling, guest-level scheduling failures effect other virtual machines, too. For example non-energy-aware guests can consume more energy than assigned. Thus other guests are penalized, as there is only a limited amount of energy available per scheduling period. Furthermore balancing the energy consumption of physical CPUs is not possible at all using guest-level scheduling. The virtual machines have no knowledge about the energy consumption of physical CPUs. The virtual machine monitor is the sole component which has an overview about the physical resources. It is the only place where we can prepare the monitoring of the energy consumption of all resources and principals in our virtual machine environment (physical CPUs, virtual machines and virtual CPUs in the multi-processor case). Thus only a host-level mechanism can schedule virtual machines to physical CPUs according to their real energy consumption. Therefore we require a *migration mechanism for virtual machines*. Additionally the introduction of *virtual machine suspension* to the virtualization layer can compensate the lack of energy-aware scheduling within a guest's operating system. Non expected energy consumption can be balanced by the virtual machine monitor by suspending the corresponding virtual machine.

The described design keeps the guest operating system's *fine-grained knowledge* about energy consumption using guest-level functionality and, coevally, we ensure our design is *not susceptible* to the influence of virtual machines unaware to guest-level scheduling by offering virtual machine suspension. Furthermore the *energy*

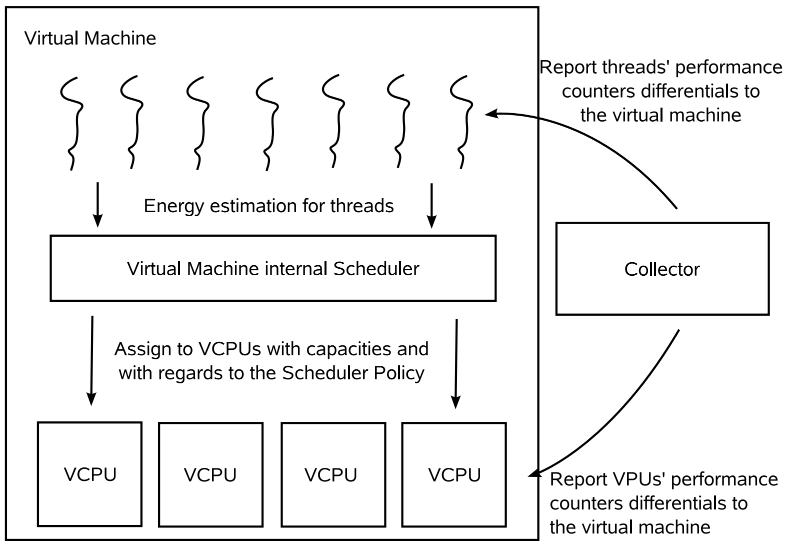
Virtual Machine
Report threads' performance counters differentials to the virtual machine
Energy estimation for threads
Virtual Machine internal Scheduler
Collector
Assign to VCPUs with capacities and with regards to the Scheduler Policy
VCPU
VCPU
VCPU
VCPU
Report VPUs' performance counters differentials to the virtual machine

Figure 3.2: Guest-level scheduling: The performance counter deviance related to threads and virtual CPUs are reported to the corresponding virtual machine which thereafter estimates their energy consumption. The guest's scheduler afterwards suspends threads in case its energy budget is exhausted. If another virtual CPU still has energy capacities left it can migrate a thread to the corresponding virtual CPU.

capacities of all physical CPUs can be exhausted using the mechanism of virtual machine migration. In the following we consider the main functionality of guest-level scheduling and host-level scheduling.

3.3 Guest-Level Scheduling

For this thesis guest-level scheduling denotes the scheduling of threads to virtual CPUs with the background of energy-awareness of the guest operating system. As can be seen in Figure 3.2 we migrate threads due to available energy capacities to the virtual CPUs to balance the energy consumption of the virtual CPUs. Furthermore guest-level scheduling provides the capability of suspension by controlling its threads' energy consumption. We use guest-level suspension to keep energy budgets, for example not to exceed temperature limits [12]. Guest-level scheduling allows to benefit from the fine-granular scheduling and from energy-related information available only within the guest operating system. Thereby guest-level scheduling achieves more exact scheduling results compared to host-level scheduling. In the following we give an overview to the components required for guest-level scheduling.

We rely on energy consumption of threads and of virtual CPUs as a decision criterion for guest-level scheduling. Therefore we utilize energy profiles. Energy profiles offer the energy consumption of an entity (scheduling resources and principals) observed for some past scheduling periods [13]. Energy profiles require an approach to measure the energy consumption for an observed scheduling entity (the scheduling resources and the scheduling principals). We selected energy estimation from performance counters as approach [5] to measure energy consumption. But a virtual machine environment relying on this approach requires further considerations. We need to virtualize the performance counters in order to measure the energy consumption of threads. Additionally we need a method for gathering the energy consumption for host-level entities (for example physical CPUs, virtual machine...). The virtualization of the performance counters is driven by a component executing in the virtual machine monitor called the collector. It also provides the energy measurement for virtual CPUs, physical CPUs and the virtual machines.

By using energy estimation guest-level scheduling can obtain the energy consumption of threads. As next step it requires a facility to account this energy consumption to its related thread. An approach to account energy consumption to its related thread is described in [16]. This approach leverages resource containers [1] for accounting of the energy consumption. To control energy consumption by the virtual machine monitor it requires a flexible mechanism which can be used to set energy budgets for virtual machines. We assign to each virtual machine a local available energy budget which can be updated from scheduling period to scheduling period. The guest-level scheduler employs furthermore thread suspension and thread migration in order to implement energy-aware scheduling policies.

In the following sections we discuss all major topics in more detail: The energy profiles, the collector, energy budgets, thread suspension and thread migration.

3.3.1 Virtual Energy Profiles

Classic operating systems do not support the measurement of any energy consumption. Thus they are not capable of behaving energy-aware. To bridge this gap we leverage *energy profiles*. Energy profiles [13] offer the monitoring of energy consumption of an entity and the prediction of the energy consumption from those monitored values. Guest-level scheduling entities to consider are the threads of virtual machines and the virtual CPUs of virtual machines. They are the relevant entities for our guest scheduler policy. If we want to offer energy profiles we have to discuss the source of the energy values. For this purpose we firstly have to consider how energy consumption can be virtualized.

Virtualizing Energy Consumption

We leverage virtual energy profiles to monitor the energy consumption of the different resources and principals of the virtual machine environment. To determine energy consumption on operating system level, we reuse the mechanism of event-based energy estimation as proposed in [5]. We shortly discussed this topic in Section 2.2.1. Leveraging this approach for a virtual machine environment additionally requires to *virtualize energy consumption*. The classic approach assumes there is only one operating system running. In our case there might be running several of them at the same time. Virtualizing energy is to divide out the energy consumption of other virtual machines running in parallel to the considered one. Virtualizing energy consumption can be done by the virtual machine monitor. It has access to the performance counters and knows about the virtual machines and when they run. Accessing and virtualizing the performance counters is discussed in the next subsection.

To distribute the energy consumption to its threads is the task of the guest-level scheduler itself. It has to assign the energy consumption to its threads for itself as the collector does not know about the threads of the guests at all. However, the collector can provide the guests with either the performance counters deviance or directly with the energy consumption.

In the following we show how to virtualize performance counters. Afterwards we discuss the collector which is the component to virtualize energy consumption for host-level entities.

Virtualizing Performance Counters

Performance counters monitor the occurrence of different performance-related events, for example unhalted cycles, and store them in related counter registers [17]. We start with host-level monitoring. In order to *virtualize performance counters* we require the counters in a place which is capable to assign the results to the considered resources and principals, virtual and physical CPUs as well as virtual machines. This component reads the counters on each context switch and thereafter assigns these values to all affected entities. For example if virtual CPU 2 of a virtual machine is assigned to physical CPU 1 the performance counters are accounted on context switch to the virtual machine, the virtual CPU 2 and the physical CPU 1.

The guest-level's principal - threads - and its resource - virtual CPUs - have to be considered separately. The virtual machine monitor generally has no knowledge about a guest's threads. Thus each guest needs to assign the energy consumption to threads by itself. We accommodate for this fact by passing the deviance of the performance counters directly to the guest. But we only pass those which occurred when the considered guest was running. In the following the guest can assign the estimated energy to the threads. Thereby the guest is furthermore capable of determining the energy consumption of virtual CPUs. The guest knows which thread is running on which virtual CPU. In Chapter 4 we will discuss the prototype. Then we will return to the virtualization of the performance counters as the further proceeding is an implementation dependent one.

Collector

We have considered how performance counters are virtualized in order to gain virtualized energy, additionally. Having virtualized performance counters we still

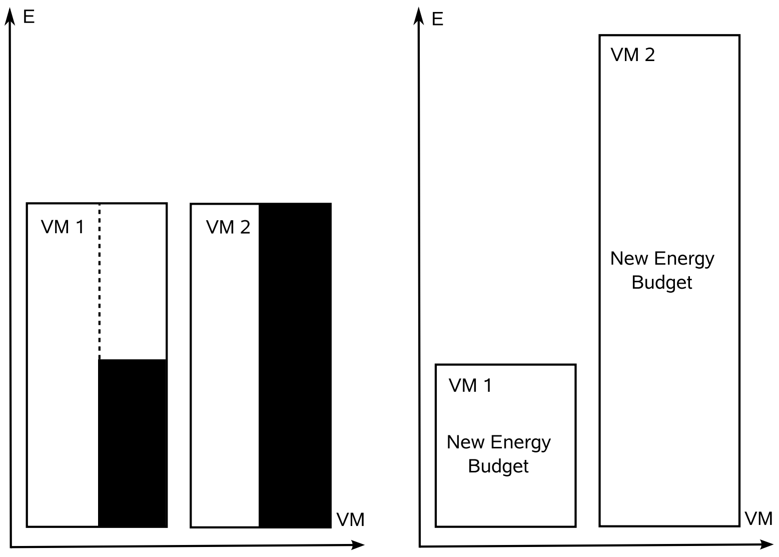

Figure 3.3: Recalibration of energy budgets: Energy budgets allow to limit the energy consumption of a virtual machine and its virtual CPUs. In order to react on consumed energy (the black area in the figure) we can recalibrate the energy budgets for the next scheduling period due to the lower energy consumption in the last scheduling period of virtual machine 1.

require the component performing the virtualization of the energy, the assignment of the virtualized performance counters and the virtualized energy to the correct scheduling entity, the maintaining of a database for all scheduling entities and the offering of an interface to get the database entries. In the following we introduce this component that we call the *collector*. The collector virtualizes the performance counters as described above. These virtualized performance counters have to be shared with the guests. For example this can be realized by providing shared memory which is filled with the deviance of the performance counters. As discussed above the guest performs the assignment of the estimated energy for itself.

Afterwards the collector estimates the consumed energy for considered host-level entities (for example virtual machines or physical CPUs). The energy values now are stored in the database separately for each involved entity. Finally the collector offers an interface that can be used to read a required energy value. If the virtual machine monitor now requires the energy consumption of the virtual machine, the physical CPUs or the virtual CPUs it simply queries the offered interface and receives the energy consumed of this scheduling entity since the last query in return. Using the collector on host-level is addressed again in Section 3.4.1 in more detail.

3.3.2 Energy Budgets

A virtual machine is regarded as a guest in the virtual machine environment. In order to schedule energy-aware a guest requires limits considering its energy consumption so it can decide when to migrate a thread or when to suspend it. In the context of energy-aware scheduling this means to define *energy budgets*, limits which have to be kept by the virtual machine for itself. Therefore we assign an energy budget to each virtual machine per virtual CPU. It is assigned by the virtual machine monitor. Thereby the virtual machine monitor can actively engage to the scheduling process within the guest. It can instruct a guest to suspend in case the energy budget is exhausted on all virtual CPUs or to migrate threads in case the energy budget is only exhausted on some virtual CPUs.

Recalibration

Energy budgets require to be *recalibrated* regularly in order to support solving overload situations in certain virtual machines or simply to better exhaust energy capacities of available CPUs in multiprocessor systems. Recalibration means to redefine the energy budgets to new values by the virtual machine scheduler because of the energy which has been consumed by a guest within the last scheduling period.

The energy consumption of all guests is only known to the virtual machine scheduler. In order to support a better exhaustion of energy capacities already on guest-level, the recalibration mechanism is provided by the guest-level scheduler. It gives the virtual machine monitor the possibility to better engage to the scheduling process of its guests and thereby participate from the benefits of guest-level scheduling.

For example, let us consider Figure 3.3 which explains the necessity of recalibratable energy budgets for a guest in order to avoid the waste of energy. A virtual machine is consuming less energy than assigned. If a recalibration mechanism is available the energy budgets can be corrected for the next schedule period and less energy is wasted. The other virtual machines can be assigned a higher budget. Thereby we also can react on overload situations in sole virtual machines. Virtual machines expecting an overload situation can be assigned more energy which is taken from one with a lower energy profile.

3.3.3 Suspension and Migration

So far we considered the required facilities in order to measure energy consumption and to define energy budgets to limit the energy consumption of virtual machines. However, there are no mechanisms which support a reaction on these budgets. Basically we can react in two different ways if the energy budget for a virtual CPU is exceeded. A virtual machine is suspended if all its threads running on the same virtual CPU have exhausted their energy capacities. Furthermore we can migrate

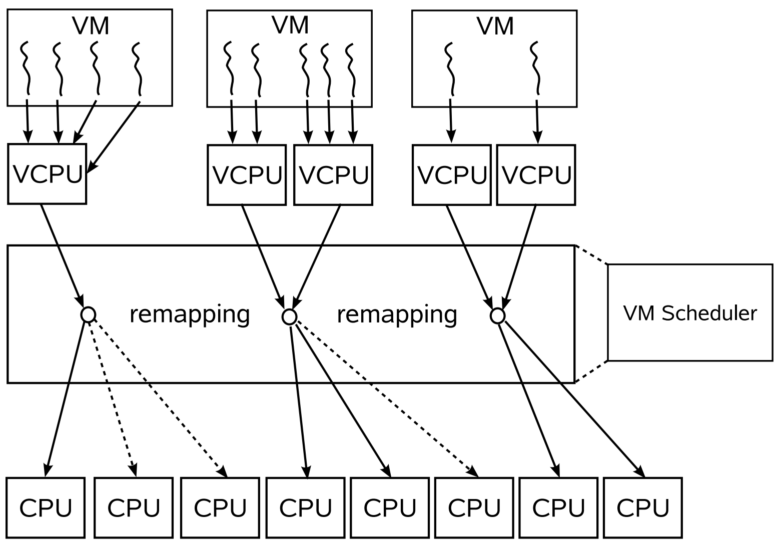

Figure 3.4: Host-level scheduling: The virtual machines internally have assigned threads to virtual CPUs. The virtual machine scheduler however controls the assignment of the virtual machines' virtual CPUs to physical CPUs. By reordering them it can migrate virtual machines between different physical CPUs.

such threads to another virtual CPU still having energy capacities available. Therefore the guest-level scheduler needs not only to monitor the energy consumption of its threads as discussed in Section 3.3.1 but it also needs to monitor the energy consumption of its virtual CPUs. This yet is not really a problem as the guest operating system knows which thread is running on which virtual CPU. Summing up their energy consumption yields the energy consumption of the virtual CPUs.

Suspension of threads in virtual machines due to energy budgets already has been discussed in [16]. This approach utilizes a power control mechanism based upon resource containers introduced in [25]. We can easily reuse this approach for suspension in guest-level scheduling.

Furthermore we propose to migrate threads to improve the exhaustion of the energy budgets of different virtual CPUs. If the energy budget of one virtual CPU is consumed completely the guest-level scheduler migrates a thread to another virtual CPU. Thus a better exhaustion of energy capacities can be achieved. Migration is a mechanism basically known to modern operating systems. When we present our prototype in Chapter 4 we will present how the standard Linux-scheduler can be extended to migrate energy-aware.

3.4 Host-Level Scheduling

In the last section we discussed the guest-level scheduling located in a virtual machine in detail. In the following we discuss its complementary part residing in the virtual machine monitor called host-level scheduling. Host-level scheduling provides all required facilities to schedule complete virtual machines by the virtual machine monitor's scheduler. In contrary to guest-level scheduling we do not control virtual CPUs, we directly consider physical CPUs.

Host-level scheduling is designed not only to be an alternative to guest-level scheduling. It is rather complementary to guest-level scheduling in different ways. On host-level we have access to informations not available to the guest operating system. For example we may read the temperature of physical CPUs. Also the virtual machine only aggregates informations concerning the energy consumption of all virtual machines. Thus enforcement of a global energy policy has to be managed by the virtual machine monitor. Furthermore host-scheduling offers a possibility to observe each virtual machine's energy consumption. In case a virtual machine is not capable of energy-aware scheduling or it is malfunctioning the virtual machine monitor's scheduler may interfere and enforce an assigned energy budget. Once again we argue guest-level scheduling and host-level scheduling are two complementary partners rather than being an alternative which is illustrated in Figure 3.4. In the following we give a component-wise overview to host-level scheduling.

Host-level scheduling requires energy profiles, especially about physical CPUs and virtual machines (in the multiprocessor case for virtual CPUs, too). For example a scheduling policy can consider to limit the energy consumption on a physical CPU or the energy consumption of a virtual machine (per virtual CPU in the multiprocessor case). Therefore we may reuse the database which is maintained by the collector. The virtual machine scheduler implements the scheduling policy. It relies on the energy profiles and on the energy budgets for the considered entities. The virtual machine scheduler controls the guest-level scheduler by invoking the recalibration mechanism for energy budgets of the guest-level scheduler as discussed in section 3.3.2. Using this facility the host-level scheduler is capable of distributing energy for physical CPUs to virtual machines and virtual CPUs. However, complete virtual machines are the first-class entities the host-level scheduler is handling. Therefore host-level scheduling offers energy budgets for virtual machines. Energy budgets for virtual machines can be assigned either statically or dynamically. Dynamic assignments support more complex schedulers requiring periodically updates of energy budgets while static assignments reduce complexity and energy overhead induced by host-scheduling. For example static budgets are suitable for scheduling policies just offering a fair energy distribution as a sole initial energy distribution is sufficient here. In order to implement energy-aware scheduling policies in the virtual machine monitor we need two further basic facilities. Firstly we use virtual machine suspension which can be used in combination with energy budgets to enforce these budgets. Secondly we use virtual machine migration. Together with energy budgets we can implement energy-aware balancing algorithms on host-level.

Below we discuss host-level scheduling in further detail and show the assembly of the major components.

3.4.1 Virtual Machine Scheduler

The *virtual machine scheduler* resides in the virtual machine monitor. It implements a scheduling policy which assigns virtual machines to physical CPUs and distributes energy budgets, we have addressed in guest-level scheduling. The decision criterion of the virtual machine scheduler are the energy profiles of virtual machines (and their virtual CPUs in the multiprocessor case) and physical CPUs.

The energy profiles are made available by the collector, which we introduced in section 3.3.1. It has access to the energy consumption of the virtual machines and physical CPUs. The host scheduler invokes a function offered by collector to query the energy profiles of the virtual machines and of the physical CPUs.

To implement its policy the scheduler requires the same mechanisms as the guest-level scheduler: Energy budgets, suspension and migration. We discuss these mechanisms in the sections below.

Energy Profiles of Virtual Machines

In section 3.3.1 the energy profiles were introduced in detail. We presented the collector as the component gathering them. Now we describe how the energy consumption of virtual machines can be detected in more detail.

Creating the *energy profiles of virtual machines* means to calculate the energy consumption of each virtual machine by dividing out the energy consumption the other virtual machines have consumed. As we use energy estimation this can be achieved by reading the performance counters on each virtual machine switch. From this we get the deviance of the performance counters for the scheduling period a virtual machine has been running. Afterwards the collector can estimate the energy consumption based on these values. Further conclusion to the energy consumption of the virtual CPUs (in case we handle multiprocessor virtual machines) and the physical CPUs can be made. We monitor the performance counter deviances per virtual machine and per physical CPU. Thereby we have the energy consumption per physical CPU, too. As the virtual machine monitor knows about the virtual CPU-to-physical CPU mapping the collector furthermore can calculate the energy consumption per virtual CPU.

Using the Collector

The collector component represents the central database offering the energy profiles. As we need to query the collector for the profiles we have to provide a corresponding function offering the access to the database entries as can be seen in Figure 3.5. Components residing in the virtual machine monitor itself (in this case the virtual machine scheduler) normally can not invoke interfaces which are defined to be used with virtual machines.

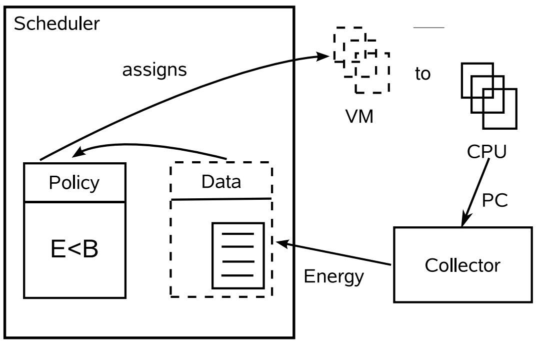

Figure 3.5: Reusing the collector: The virtual machine scheduler reuses the database maintained by the collector. It monitors the energy consumptions of virtual machines and physical CPUs. After querying the energy the scheduler can decide dependent on its policy which virtual machine needs to be migrated to which CPU or which virtual machine better is to be suspended.

3.4.2 Controlling the Guest-Level Scheduler

The guest-level scheduler's energy budget controls the energy consumption of threads on a guest's virtual CPUs and thus the energy consumption of the complete virtual machine. A preassigned energy budget is kept per virtual CPU. These budgets can be defined by the scheduler of the virtual machine monitor. By offering this mechanism the scheduler can actively participate from the advantages of guest-level scheduling, for example from its higher preciseness. An appropriate virtual machine scheduler can be reduced to analyze and distribute the available energy on the physical CPUs which the virtual CPUs of the current virtual machine are assigned to. The combination of guest-level mechanisms and those we are introducing below however often is inevitable, for example if guest-level schedulers do not support energy-aware scheduling.

3.4.3 Energy Budgets on Host-Level

The virtual machine scheduler can rely on the guest-level mechanisms as addressed above. But the scheduler still lacks of a mechanism to exhaust all energy capacities available, for example if the virtual machines initially are not running on all physical CPUs. The energy capacities on the unused CPUs will never be touched. Furthermore it was not capable to enforce the assigned energy budgets of the guests. Therefore we have to offer mechanisms on host-level scheduling handling those issues. Host-level scheduling principals are virtual machines running on physical CPUs. Thus at first we require energy budgets for virtual machines (once again per virtual CPU to support the multiprocessor virtual machines) and for physical CPUs.

Static Energy Budgets

Energy budgets are a mechanism to divide available energy capacities on the physical CPUs between all virtual machines on host-level. The virtual machine scheduler assigns these energy budgets. *Static energy budgets* are the first model available, offered due to its lower complexity and the lower energy overhead. The energy budgets are assigned on start-up and will not be changed anymore. This is enough for scheduling policies which want to achieve fairness between all virtual machines or simply require only a fixed distribution.

But especially those scheduling policies which want to achieve fairness between all virtual machines better rely on guest-level functionality. As we will demonstrate in Chapter 5 guest-level mechanisms achieve a higher preciseness in keeping their energy budgets and a more even distribution of energy between threads.

Dynamic Energy Budgets

Static energy budgets however are not sufficient if fairness between virtual machines or physical CPUs is not the only goal. In particular for the achievement of an optimized workload in our system it is essential to be able to *dynamically reassign energy budgets* after each scheduling period in a similar way recalibration offers reassigning budgets for the guest-level scheduler's scheduling entities (see section 3.3.2). A potential virtual machine scheduler requires to react on different workloads of a virtual machine. It needs to reassign lower energy budgets if the virtual machine did not exhaust its energy budget completely; otherwise it has to assign higher budgets if a virtual machine reached its limit.

Resource allocation strategies utilizing this mechanism can leverage it to define a global scheduling policy independent of the scheduling policy of the guest's scheduler. For example temperature-aware scheduling (see section 2.2.3) can benefit from dynamic energy budgets. The scheduler can assign energy budgets deduced from the temperature of physical CPUs to virtual machines which are currently running. At first glance guest-level scheduling can be adducted for this task. The problem is that the guest-level scheduler can not be forced to support energy-aware scheduling. In this case a single virtual machine can undermine the policy of our system by consuming as much energy as possible. Balancing the temperature obviously is not possible then.

3.4.4 Virtual Machine Suspension

To this point we discussed the basic control mechanism for energy-aware scheduling: Energy budgets for virtual machines per virtual CPU and physical CPUs. Energy budgets offer the possibility for virtual machine scheduler to control the energy consumption of the virtual machines. However, it requires further operations in order to react to the reaching of limits. As first mechanism we discuss *suspension of virtual machines*.

As addressed above guest operating systems can not be forced to operate with energy-aware scheduling. Wrong configuration on compile-time, non-availability or simply malfunctioning because of implementation issues risk potential damages to the underlying hardware. Besides hardware issues suspension is justified by the fact that energy illegally consumed by a non-energy-aware virtual machine reduces the energy capacities available to other virtual machines. Both issues call for the control of virtual machines by the virtual machine scheduler.

Suspension of a virtual machine is a possible solution for both problems. A virtual machine can be halted for the rest of a scheduling period if it has consumed its assigned energy budget. The reaction time of virtual machine suspension must be kept rather low to avoid impreciseness. Therefore, depending on the final virtual machine scheduling algorithm, a trade-off between impreciseness and further induced energy overhead has to be made. To return to the example of temperature-aware scheduling, the CPU temperature is a rather slowly changing value in service [12]. Once a CPU reaches its working temperature the latter changes slowly within multiple seconds even on high workload. Thus reaction times within a second can be considered adequate.

3.4.5 Virtual Machine Migration

Virtual machine suspension addresses issues related to guests disregarding the virtual machine scheduler's energy distribution. However, we also require a mechanism to explore energy capacities available on CPUs no virtual machine is assigned to. Otherwise those capacities were definitely lost. For example two virtual machines share the physical CPU 0. If the energy capacities of this CPU are exhausted they are suspended. A better alternative is to assign them to the physical CPU 1 to avoid suspension and exhaust these additional energy capacities. Figure 3.6 illustrates the benefits of virtual machine migration for a 2 CPU system.

Virtual machine migration as demonstrated in [6] can be consulted as an approach to overcome this issue. A virtual machine is completely migrated between different physical CPUs. Thereby we are capable to exhaust the energy capacities available on unused physical CPUs.

Once again we have to consider a trade-off, in this case between the exhausting of energy capacities on the one hand and the energy overhead and performance loss induced by the migration process itself on the other hand. A generalized answer to this topic is difficult as it depends on the implementation of the participating components. But, as we will demonstrate in Chapter 5, we can implement migration of virtual machines with a low energy overhead. Thus virtual machine migration offers a possibility to benefit from energy capacities on unused CPUs.

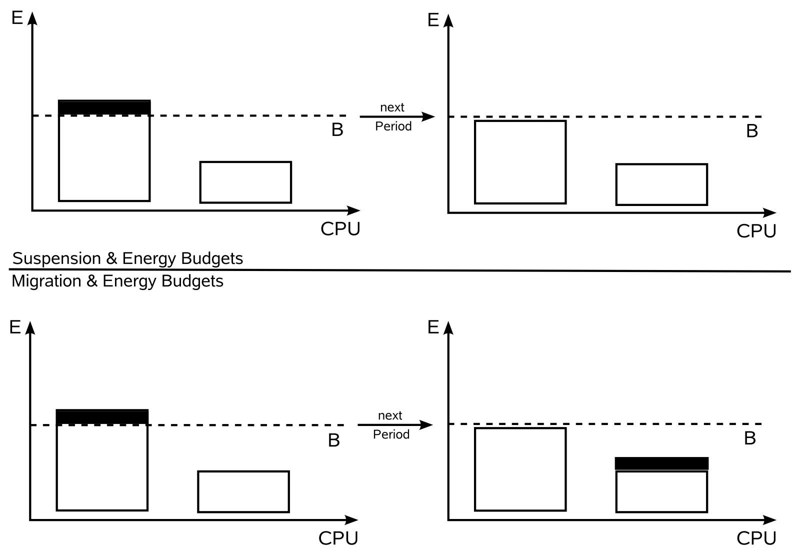

Figure 3.6: Migration vs. suspension of virtual machines: The scheduler can implement host-level scheduling policies relying on suspension and migration. Different policies like energy-aware scheduling or throttling can be realized quickly using this facilities.

3.5 Summary

We introduced scheduling mechanism on both levels of a virtual machine environment. Guest-level scheduling offers mechanisms which admit the guest's scheduler to keep energy budgets on its virtual CPUs. The guest's scheduler can employ energy budgets, thread suspension and thread migration, and offers a recalibration mechanism for its energy budgets to the virtual machine monitor. Host-level scheduling consists of a collection of mechanisms which can be used for the implementation of scheduling policies having an energy background only measurable or controllable by the virtual machine monitor. The virtual machine scheduler can rely on static/dynamic energy budgets, virtual machine suspension and virtual machine migration. Furthermore it can actively interfere to the guest's scheduling process if it supports guest-level scheduling by using the recalibration mechanism of a guest's energy budget.

Initially we presented four design goals we claimed to follow. To finalize this section we present their achievements here.

- **Accuracy.** At first we mentioned accuracy. Accuracy is closely dependent to the implementation of our design. Because of the splitting into guest-level and host-level our approach offers high accuracy if required. A scheduler requiring high accuracy can rely on guest-level scheduling as far as possible (see section 3.3 for reasoning).

- **Safety.** Second we mentioned safety in the context of non-energy-aware virtual machines. We solved this problem by introducing host-level suspension. It can be used to enforce energy budgets by the virtual machine monitor (see section 3.4.4.).

- **Flexibility.** Third we addressed flexibility. The developer of the scheduling policy is capable to decide which mechanisms must be used depending on the goals the scheduling policy wants to achieve. As the scheduler can be implemented in the guest, in the host or in both components, flexibility is granted, too.
- **Efficiency.** Fourth we requested efficiency. The efficiency of our approach is given as we add only a minimal set of new functionality to a virtual machine environment required for energy-aware-scheduling (see section 2.2.3). Further proof for the efficiency will be given in Chapter 5 when we evaluate our prototype.

Chapter 4

Implementation

We have implemented energy-aware scheduling support for virtual machines in a multiprocessor environment for an existing virtualization environment: *The L4Ka virtualization environment* developed at the University of Karlsruhe. Implemented upon a fast micro-kernel resides the L4Ka para-virtualization layer. It offers all common facilities like virtual CPUs or guest isolation [23]. The virtualization layer also includes network support and disk support by using an approach for device driver reuse [9]. Multiprocessor support basically is given, too.

We implemented a prototype of guest-level schedulers and host-level schedulers supporting the described platform. Guest-level scheduling is capable to schedule threads to multiple virtual CPUs in an energy-aware manner. It dynamically reacts on changes of the assigned energy budgets. The virtual machine scheduler has been extended to support host-level scheduling. We completely virtualized the performance counters, the energy consumption and implemented the collector. Support for energy profiles is given for all essential entities on each level. We also implemented an effective facility to distribute energy budgets. Virtual machine suspension and virtual machine migration is implemented, too. For a better evaluation we further implemented two energy-aware scheduling policies: Energy-aware balancing and throttling for both guest-and host-level scheduling.

4.1 Overview

In this section we present the design's components we implemented for our prototype. We begin by discussing the implementation of guest-level scheduling. We present the realization of energy budgets and an efficient way to recalibrate them. We show how we implemented the virtualization of the performance counters and the energy consumption. We discuss the collector responsible for the calculation of the energy profiles for the different scheduling entities. Afterwards we outline the energy budgets. Then we discuss guest-level suspension and guest-level migration. Finally we shortly address two energy aware scheduling policies: Energy-aware balancing and energy-aware throttling implemented for guest-level scheduling.

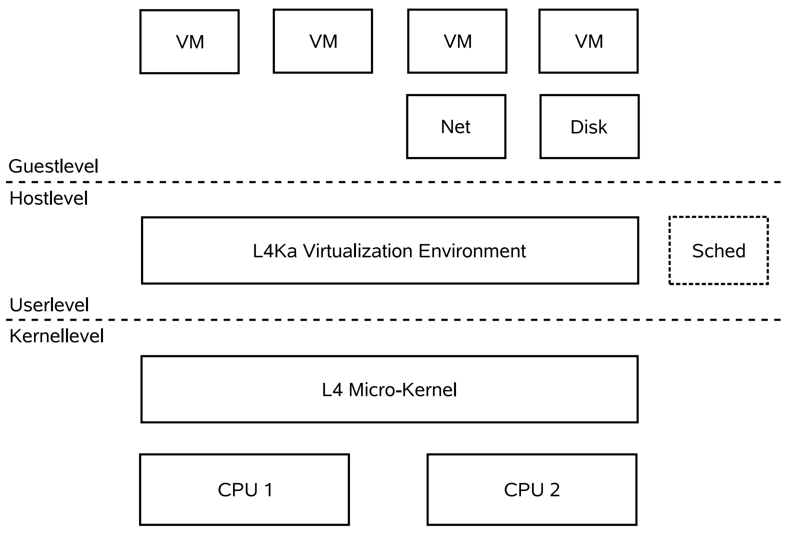

Figure 4.1: Environment: Upon a Pentium D, 2x3GHz, a L4 micro-kernel driven virtualization environment hosts a various number of guest virtual machines.

Thereafter we consider the components of host-level scheduling. As it is a basic component we begin by discussing the virtual machine scheduler and by doing so we show how to reuse the collector. Then we proceed with the implementation of the energy budgets for virtual machines. A major part takes the implementation of virtual machine migration and virtual machine suspension. Finally we show how we implemented energy balancing and energy throttling in the virtual machine scheduler.

4.2 Environment

Our prototype is implemented for the L4Ka virtualization environment developed at the System Architecture Group of the University of Karlsruhe (see Figure 4.1). In contrast to other virtual machine monitors like Xen [2], our virtual machine monitor is an unprivileged user-level instance which is running on top of a micro-kernel. The used micro-kernel is the *L4 micro-kernel* [10,20], developed also at the University of Karlsruhe. The L4Ka virtualization environment does not provide an own scheduler. Instead it relies on the scheduler implemented in the L4 micro-kernel. The L4 scheduler implements a priority-driven round robin scheduler. It additionally offers timeslices to control thread preemption. By a clearly specified interface it allows the implementation of *user-level schedulers* by modifying the schedule parameters of associated threads. However, we provide a framework which abstracts from this interface to offer a better comprehension of the ongoing processes.

The guest operating systems are Linux 2.6 called *L4Linux* [7], which are para-virtualized kernels so they cooperate with the L4 virtualization architecture. The environment provides device drivers for network and disk applying the technology of unmodified device driver reuse [9]. These drivers are running in dedicated virtual machines executing a L4Linux instance which has pure access to the related hardware. Furthermore it offers a server which can be accessed by a specified interface. Client virtual machines install a client driver which can communicate with the server and perform actions by invoking this interface. Communication between virtual machines and the virtual machine monitor is implemented as hypercalls. These calls reuse the IPC mechanism offered by the L4 micro-kernel to transfer information. They are fast enough to create only an insignificant overhead on communication [22].

4.3 Guest-Level Scheduling

Guest-level scheduling is concerned with scheduling threads to virtual CPUs in an energy-aware manner. Virtualized energy is the basis for our energy accounting facility. We reused *resource containers* [1] for the accounting of energy consumption. We assign energy consumption to the thread which originally caused it. Energy budgets offer a flexible mechanism to limit the consumption of energy on thread-level. The approach of [25] which provides resource accounting and controlling is applied with modifications introduced in [16] (see section 2.2.2). In addition we outline the way we support communication between the different virtual machine components (for example to transfer energy budgets). Virtual machine environments offer the possibility to share informations by using *synchronous IPCs or shared memory*. We discuss both approaches and reason our applied selection. Afterwards we detailed engage in suspension and migration. We conclude this section by shortly introducing two implemented energy-aware scheduling policies: *Energy-aware throttling* and *energy-aware balancing*. Furthermore we demonstrate how to implement energy-aware scheduling policies straight forward upon our framework.

4.3.1 Virtualizing Energy

Virtualizing energy is a process originating in event-based energy estimation [5]. Energy estimation is based on the CPU's performance counters. By quantifying them with CPU-adjusted weights the energy consumption can be calculated. Thus, at first, we have to discuss how to obtain performance counters in a virtual machine environment, a process we call virtualizing the performance counters. This process is to divide up the performance counters as far as possible in order to assign it to all scheduling entities available in a virtual machine environment: Threads and virtual CPUs for the guest scheduler; virtual machines and physical CPUs for the virtual machine scheduler.

```
LOG_CODE(log_time, u32_t this_domain)
{
    u64_t buf64=0;
    cpu = current_cpu();

    if (LOGGING_THREAD_EXIT(LOGGING_RESOURCE_CPU, this_domain))
    {
        PC_ASM_RDTSC( buf64 ); // Time Stamp Counter
        NEW_TSC[cpu] = (unsigned int) buf64;
        dTSC = NEW_TSC[cpu] - PMC_TSC_OLD[cpu];

        PC_ASM_RDPMC( 0, buf64 ); // MSR_P4_BPU_PERFCTR0
        NEW_UC[cpu] = (unsigned int) buf64;
        dUC = NEW_UC[cpu] - PMC_UC_OLD[cpu];

        // ... continued for all further performance counters ...

        PC_ASM_RDPMC( 16, buf64 ); // MSR_P4_IQ_PERFCTR4
        NEW_UKN[cpu] = (unsigned int) buf64;
        dUKN = NEW_UKN[cpu] - PMC_UKN_OLD[cpu];

        log_counters(this_domain, dTSC, dUC, ..., dLDM);
    }

    if (LOGGING_THREAD_ENTRY(LOGGING_RESOURCE_CPU, this_domain))
    {
        PC_ASM_RDTSC( buf64 ); // Time Stamp Counter
        PMC_TSC_OLD[cpu] = (unsigned int) buf64;

        PC_ASM_RDPMC( 0, buf64 ); // MSR\_P4\_BPU\_PERFCTR0
        PMC_UC_OLD[cpu] = (unsigned int) buf64;

        // ... continued for all further performance counters ...

        PC_ASM_RDPMC( 16, buf64 ); // MSR_P4_IQ_PERFCTR4
        PMC_UKN_OLD[cpu] = (unsigned int) buf64;
    }
 }
```

Figure 4.2: Virtual performance counters: Logging calls LOG_CODE on thread_enter and on thread_exit events. On a thread_enter event we determine the current performance counters for the CPU the virtual machine is running on. On a thread_exit event we read the performance counters once again and afterwards calculate the deviance. Finally we sent the deviance of each counter to our logger.

Virtualizing Performance Counters

In this section we describe the virtualization of the performance counters. On each context switch of a virtual machines we read the performance counters and store them. *Context switches of virtual machines* take place in the L4 micro-kernel. Thus the logging of the switches has to be implemented here. An efficient approach to *log system characteristics* is introduced in [19]. We use this approach to monitor the performance counters' deviance for each virtual machine switch. As demonstrated in Figure 4.2 we read the current performance counters if a virtual machine begins to run. The logging facility sends an *thread_enter* event to inform us about this occurrence. On a *thread_exit* we can determine the performance counters once again. Thereafter we have to calculate the deviance of all performance counters. Finally we write them to the log file. When we call the *log_counters*-function we additionally pass the virtual machine identifier. The logging facility passes this identifier to our *LOG_CODE* function. When we discuss the collector we will see the support which offers this availability to determine the energy consumption for the considered host-level entities (virtual machines, physical CPUs and virtual CPUs).

The logged data however is still not available to the guest. For now they are only separated for each virtual machine. Thus we have to send them to guest. This can be implemented by sharing the logged performance counters in an aggregated way with the guest. We sum up the deviances for a considered scheduling period and then we share the aggregated values with clients. Thereby the resource allocator within the guest can continuously calculate the energy consumption and assign it to the last executed thread.

Energy of Threads

The performance counters now are virtualized and available from the shared memory page within a guests. In the following we have to estimate the energy consumption of the threads. The estimation formula is described in [5]. To perform the calculation we consult the virtual performance counter deviances we stored in the logging database. By weighting the aggregated deviances with the values from the paper above we gain the energy consumption of the thread we calculated the deviance for. By iteratively inspecting the logged data a proper function of the guest's scheduler can calculate the energy consumption of the thread which ran last.

Collector: Energy Profiles on Host-Level

On host-level the energy consumption is required at least for physical CPUs and for the virtual CPUs of each virtual machine in order to support multiprocessor virtual machines. We now discuss the collector which implements the calculation of energy consumption of physical CPUs and virtual machines. Furthermore to support *multiprocessor virtual machines* we demonstrate how we gather the energy

```
void accounting(cpu, vm)
{
    while (count < LOG_ENTRIES(c)) {

        DEC_LOG(current_idx, c);

        // read all 8 pmc for the correct domain and cpu.
        counter8 = *current_idx;

        DEC_LOG(current_idx, c);
        counter7 = *current_idx;

        ...

        DEC_LOG(current_idx, c);
        counter1 = *current_idx;

        DEC_LOG(current_idx, c);
        currentthread = *current_idx;

        // calculate energy consumption of current thread ...
        energy =  6.17 * (counter1/100) + ... + 29.96 * (counter7/100)
                  + 13.55 * (counter8/100);

        // store energy consumption of physical CPU and virtual machine
        energy_consumption_CPU[cpu] += (int) energy;
        energy_consumption_VM[vm] += (int) energy;

        // finally store the energy consumption for the virtual CPU
        vm->update_energy(cpu, energy);

    }
}
```

Figure 4.3: Energy accounting by the collector: At first we read all 8 counters from the log file. Then we estimate the energy consumption. per CPU and virtual machine. This procedure is finalized by saving the energy consumption for physical CPUs, virtual machines and virtual CPUs (in order to support multiprocessor virtual machines).

consumption of a virtual machine's virtual CPUs.

The collector at first is the connection of the virtual machine monitor to the logging facility, especially to the logged performance counters. We extract the virtualized performance counters per virtual machine and per physical CPU. Thereby we can calculate the energy consumption for each entity relevant on host-level: Virtual CPUs and physical CPUs.

Figure 4.3 shows the main function of the collector *accounting(cpu, vm)*, which is executed for each CPU and each virtual machine. We parse the logged entries and thereby we obtain the counters' deviance. For each entry the energy consumption is calculated. Now the energy consumption for the last scheduling period is available. The next step is to assign this value to all scheduling entities we have to consider. The first entity is the physical CPU. We add the energy consumption directly to a dedicated array as the logging facility observes all physical CPUs anyway. The same holds for virtual machines.

In order to support multiprocessor virtual machines finally we have to assign the energy value to the virtual CPUs of the considered virtual machine. We have written a wrapper that basically determines the virtual CPU from the physical CPU which we pass as first argument to the *update_energy(vcpu, energy)*-function. This function stores the energy in a dedicated array for the determined virtual CPU of the currently regarded virtual machine.

4.3.2 Energy Budgets

We offer energy budgets to our guest-level scheduler in order to define limits for energy consumption on virtual CPUs (see section 3.3.2). Our prototype offers the assignment of energy budgets per virtual CPU. This is realized by assigning a buffer shared between a virtual machine and the virtual machine monitor. This buffer contains parts reserved for the energy budget of each virtual CPU; it can be written by the virtual machine scheduler if it decides to assign new budgets to the virtual machine's virtual CPUs. The counterpart in the guest scheduler periodically inspects this buffer. If it encounters a new value for the energy budget of a virtual CPU it refreshes its local buffer and informs the guest-level scheduler to imply them in the next scheduling period. Thus we coevally have solved the problem of recalibrating the energy budgets. The guest-level scheduler consequently can rely on energy budgets and it offers a recalibration mechanism which can be used by virtual machine scheduler.

4.3.3 IPC vs. Shared Memory

We implemented the data flow (the setting of energy budgets) between virtual machines and virtual machine monitor using shared memory. The alternative are IPCs used to implement hypercalls.

Originally we implemented the complete data flow (setting energy budgets,

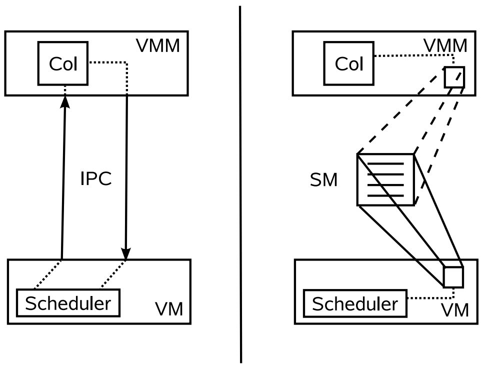

Figure 4.4: IPC vs. shared memory communication exemplified between the guest-scheduler and collector: An IPC offers a synchronous dataflow. However it consumes much energy. SM only offers an asynchronous dataflow. Therefore it requires almost no energy.

querying energy consumption, etc.) by using the IPC functionality of the L4 microkernel since it is a fast and reliable facility [22]. However, speed has its tribute as it seems. Neither the energy consumption was static nor was it low. So we had to face the problem how to deal with this variable energy overhead. A further issue arises when we use more than one virtual machine as the number of IPCs then increases, too.

So finally we decided to dedicate each virtual machine a shared memory buffer with the virtual machine monitor. We reserved an amount of this memory buffer which can be used for the dataflow of all functionality (such as setting energy budgets) as illustrated in Figure 4.4. We decided to sacrifice the advantage of synchronism (a little more precise energy accounting) offered by IPCs in order to reduce the energy overhead introduced by the virtual machine monitor. The loss of synchronism can be neglected as we are updating the shared buffer with high frequency. As we will see in the evaluation chapter the energy overhead of the virtual machine monitor using shared memory is continuously below 1 watt.

4.3.4 Suspension and Migration

To this point we discussed the fundamentals to implement the guest-level scheduling for our prototype: Energy budgets and communication. In this subsection we focus on the mechanisms required to react on energy related events, such as surpassing the energy budget.

Suspension

Guest-level suspension is derived from the approach introduced in [16]. The approach relies on a system to control resource consumption of operating systems [25]. Resource containers are applied to store the resource consumption - in our context energy consumption - for each thread in the guest system. Furthermore a mechanism is offered which ensures only such threads are scheduled which still have energy left. Therefore we monitor the energy consumption of threads using the resource container technology to account and control it. Using this mechanism a scheduler can assign each thread in a fine-granular way an energy budget which is ensured to be available. We assign this energy budget in away the energy budget of the virtual CPU is not exceeded, for example uniformly distributed. Thus we gain a functionality controlling the energy behavior of threads and coevally the energy behavior of the virtual CPUs by controlling the energy consumption threads.

In case we found the energy consumption of threads running on a considered virtual CPU is beyond the virtual CPU's energy budget we additionally can prevent it from scheduling any further threads to this virtual CPU until the energy budget of this virtual CPU is refreshed again. The described functionality offers virtual machine suspension per virtual CPU by keeping energy budgets on thread granularity.

Migration

Guest-level migration allows to assign threads to other (virtual) CPUs. Our guest operating system, L4Linux 2.6, offers migration of threads between different CPUs in well a tested way. We demonstrate how the standard Linux scheduler can be extended for migrating energy-aware instead of due to the CPU load.

In a first step we have to modify the Linux scheduler to install energy as the migration criterion. This can be done in the *rebalance_tick*-function where the runqueue workload of the current CPU is updated. Instead of updating the current runqueue with the new CPU load we update it with the energy consumption of the current CPU. Thus afterwards the Linux scheduler uses energy consumption of threads as decision criterion.

After some testing however we found two problems: At first the energy consumption has very high fluctuations. Thus the calculated energy consumption of virtual CPUs directly is not the best indicator. Second we want to have a sole function where the energy-aware scheduling policy itself can be implemented. Both issues required further changes. In our implementation we do not assign the energy consumption of CPUs directly. Rather we assign an imaginary high or low value which ensures migration from or to this CPU is performed by the Linux scheduler. This offered a further opportunity: We can merge the determination of the energy characteristics and the mechanism for migration (and suspension) to one fixed point in our code. Thus we provide a sole point to implement the scheduling policy requiring only a little knowledge about the implementation of our prototype. We implemented high abstractions such as *set_cpu(vcpu)* to set the target virtual CPU for the thread migration logic, *no_energy_available()* to be capable to additionally halt the virtual machine in order to increase the preciseness and *energy_available_cpu(vcpu)* to decide if energy budgets are exhausted on a virtual CPU or capacities are still available.

```
void energy_aware_throttling()
{
    while (1) {

        if ( !energy_available_cpu(0) || ... ||
             !energy_available_cpu(n) ) {

             no_energy_available();

        }
    }
}
```

Figure 4.5: Energy-aware throttling: Using the proposed functionality its a straight forward and fast task to implement energy-aware policies.

set_cpu(vcpu) receives the virtual CPU where we migrate threads to if rebalance_tick() called next. It is implemented by setting the work load of the considered virtual CPU to a low value while setting the other virtual CPUs' work load to a very high value. Therefore the scheduler has the possibility to downscale eventually appearing fluctuation of the energy consumptions before assigning the work loads, for example by calculating the exponential average of the energy consumption. *no_energy_available()* halts the execution of threads for a scheduling period, for example to ensure the energy budgets of virtual CPUs are kept. At the moment our prototype can not halt a single virtual CPU. It rather halts the complete virtual machine. We implemented the mechanism by setting a variable which tells the Linux scheduler not to schedule any further threads. It is considered as additional mechanism to overcome the impreciseness of the resource container based mechanism we normally use for accounting and controlling the energy consumption of threads. *energy_available_cpu(vcpu)* receives the considered virtual CPU as parameter and returns if energy is still available. We implemented this function by parsing the shared memory page for the energy budget for the requested virtual CPU. Afterwards we compare it to the energy consumption which happened to the very moment on this virtual CPU.

Having those mechanisms policies for guest-level scheduling can be implemented. We demonstrate this in the next section by implementing guest-level throttling and guest-level balancing.

4.4 Guest-Level Policies: Throttling and Balancing

We elaborated mechanisms which are required to implement an energy-aware scheduling policy. Now we demonstrate their efficiency in order to implement a policy upon those mechanisms. As examples we consider energy-aware throttling and energy-aware balancing.

4.4.1 Energy-Aware Throttling

Energy-aware throttling is a straight forward policy which stops the execution of threads if the energy budget for a virtual machine is exhausted. Our prototype is not capable of doing so per virtual CPU. An extension to this can be easily made by extending the *no_energy_available()*-function. The basic energy-aware throttling algorithm as illustrated in Figure 4.5 continuously probes if there is still energy available on all virtual CPUs. If not it halts the virtual machine. Energy-aware throttling is not very efficient as we do not exhaust the limits of virtual CPUs. However, the claimed self-control of its energy budget is given to each guest. In order to better exhaust the energy capacities of virtual CPUs we now consider energy-aware balancing.

4.4.2 Energy-Aware Balancing

Energy-aware balancing ensures the energy consumption on all virtual CPUs is the same. On each scheduling period it controls if the consumed energy on available CPUs is significantly different. In case the difference passes a predefined threshold it sets a new virtual CPU where threads have to be migrated to. This is done by invoking the function *set_cpu(vcpu)*. By doing so iteratively energy consumption on all virtual CPUs is balanced.

This algorithm does not care at all about migration side effects, like CPU affinity. However, as we use the Linux scheduler, major side effects of migration are considered by the Linux scheduler for itself. For a better implementation one only has to extend our functionality by getting some further informations from the Linux scheduler, like the cache affinity of threads which have to be migrated. Getting further details about threads indeed is no problem as we implemented the policy within the guest's scheduler. Furthermore averaging the energy consumption of the virtual CPUs over a period of time can prevent threads to be migrated in a ping-pong fashion.

4.5 Host-Level Scheduling

In the following we discuss the major parts of host-level scheduling, our approach to support global energy-aware scheduling policies and to control non-energy-aware guests. At first we have to discuss the structure of the virtual machine scheduler. The virtual machine scheduler is the point where the host-level scheduling policy is implemented. Thereafter we discuss how to reuse the collector and how to define energy budgets for virtual machines. The major host-level mechanisms, suspension and migration finalize the features of our prototype. In order to fortify our implementation we finalize the chapter by shortly discussing energy-aware throttling and balancing as example policies.

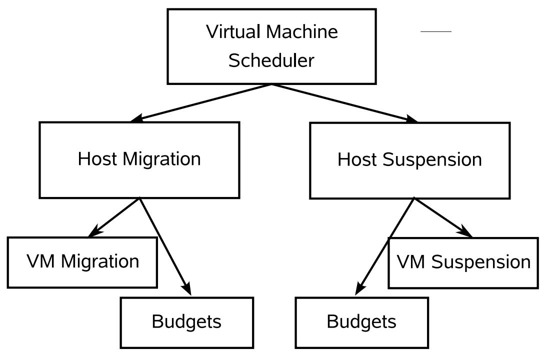

Figure 4.6: Virtual machine scheduler: The scheduling process is split up into two processes: Host-level migration and host-level suspension, each relying on its corresponding mechanisms

4.5.1 Virtual Machine Scheduler

In this section we extend the virtual machine scheduler to supply the functionality required to schedule virtual machines to physical CPUs due to energy budgets. Its final actions depend on the policies the scheduler has implemented. As Figure 4.6 demonstrates we decided for our prototype to split up the decision process into two subprocesses. The first one controls the migration related tasks and the second one the suspension related tasks. Thereby we achieve the possibility to handle policies in the same way as we did in guest-level scheduling. The process of synchronizing both policies thereby gains transparency. Furthermore we can define independent policies for each part. This is a point of interest if a policy requires only one of both mechanisms. For example, if we only use host-level migration in order to exhaust energy capacities of all physical CPUs, we can deactivate host-level suspension. By doing so additionally the energy consumption of the virtual machine monitor is reduced.

The procedure itself is basically as follows. The virtual machine scheduler calls the functions for host-level migration and suspension if a violation of one of its rules considering the energy budgets is detected (see Figure 4.8). Within these two functions the policies are implemented. Both functions have access to the energy budgets and energy profiles of the related entities (virtual machines and physical CPUs) and can make use of virtual machine suspension and virtual machine migration to realize a scheduling decision. All required informations and mechanisms are available in one place. Implementing the policies thus is straight forward as we will see in section 4.6.

4.5.2 Energy Profiles and Energy Budgets

The information we require for an energy-aware scheduling policy on host-level are the energy consumption of the virtual machines and of the physical CPUs. Thus we

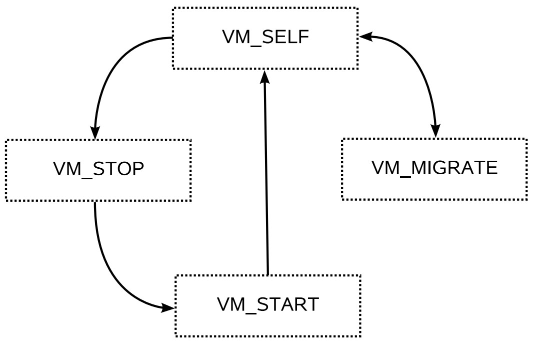

Figure 4.7: Inplace monitor state diagram: Beginning from a virtual machine state set to VM_SELF we can stop (VM_STOP), start (VM_START) and migrate (VM_MIGRATE) a virtual machine.

require the services of the collector which can provide those informations by providing the energy profiles of the virtual machines and the profiles of the physical CPUs. In section 4.3.1 we described the way we calculate the energy consumption of virtual machines and of physical CPUs. Energy profiles can be compiled easily by taking the energy consumption of an entity and afterwards calculating the average for a number of scheduling periods. For our prototype considering more than one scheduling period did not offer any advantages than directly to take the energy consumption as scheduling indicator. Thus we consider only the current scheduling period and set the energy profiles equal to the energy consumption of the current scheduling period.

We now give a hint how we organize and access the energy data. We focused on a fast and low energy consuming solution in order to keep the energy consumption induced by the virtual machine monitor as low as possible.

We store the *energy profiles for each virtual machine* in an array for a scheduling period per virtual machine and per CPU. These variables are globally available by what each access to the informations requires only negligible low energy consumption. By maintaining the information as well for virtual machines as for both CPU types (virtual/physical) the virtual machine scheduler can react in the moment a violation of a rule occurs. There is no need to post-process the data before they can be used. Low reaction latencies are the consequence.

Furthermore a scheduling policy requires energy budgets. We define energy budgets per scheduling period for each virtual machine per virtual CPU to support multiprocessor virtual machines and for physical CPUs. They are stored in an array which once again can be accessed globally for the same reasons as stated for the energy profiles.

4.5.3 Sharing the Inplace-Monitor

The L4Ka virtualization environment has a speciality concerning the virtual machine monitor. Parts of it currently reside within the address space of the virtual machine itself. The so called *inplace-monitor* deals with pagefaults and performs some of the instruction virtualization. It is created for each virtual machine one time within the boot process of the system. The inplace-monitor implements parts of the virtual machine suspension and the virtual machine migration. Because there is only one inplace-monitor per virtual machine it can only perform one of both tasks at once. A concept for sharing the inplace-monitor of a virtual machine is required. Thus we added a virtual machine state variable to each virtual machine. The virtual machine state variable indicates if the inplace-monitor is already performing one of both actions (suspension/migration). It prohibits the virtual machine monitor from the initiation of a new action if another one is already processing .

The state diagram of the inplace monitor is illustrated in Figure 4.7. It shows that *VM_SELF* is the original state. After each process the state variable is reset to this value. Only if the variable has this state the virtual machine scheduler is capable to start a new action. If *VM_STOP* is set the inplace-monitor is addressed to stop a virtual machine. *VM_START* consequently addresses it to (re-)start a virtual machine. Setting *VM_MIGRATE* indicates the inplace-monitor shall perform a virtual machine migration.

4.5.4 Virtual Machine Suspension

Virtual machine suspension is to halt a virtual machine for a period of time and afterwards restart it. We suspend a virtual machine by halting all threads to the considered virtual machine ,including the Linux kernel thread and the L4 IRQ-thread.

Monitoring Thread Creation

Creation of new threads for guests is done in the virtual machine monitor of the L4Ka virtualization environment to ensure only a limited number of threads can be created. In order to know all threads running within a certain virtual machine we monitor the event of thread creation within the virtual machine monitor.
Whenever a virtual machine creates a new thread this process is passed to the virtual machine monitor. In the following the virtual machine monitor creates the thread and returns the thread identifier back to the virtual machine. Then we store a mapping from thread to virtual machine in a dedicated buffer which we attached to each virtual machine. This buffer can be read by the virtual machine monitor by calling the function pair *get_first_dmapping(), get_next_dmapping()*. As we discuss later we furthermore require a function pair *get_first_mapping(cpu), get_next_mapping(cpu)* in order to find threads dependent on which physical CPU they are currently running.

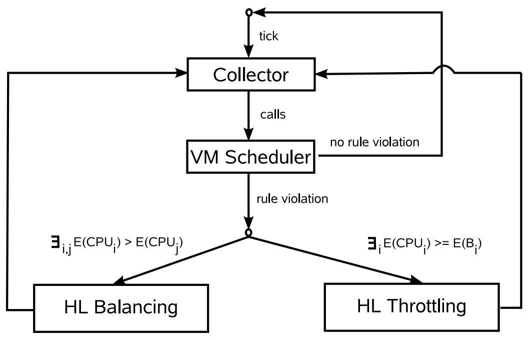

Figure 4.8: Host-level throttling and balancing: The diagram shows the design of the scheduler, the rule violations, and the separation of both policies.

Compiling the Thread List

Now the virtual machine has to be suspended. The suspension process itself is separated into two steps. At first the virtual machine monitor compiles a list containing all threads which have to be halted. We can compile this list by calling the functions described above(*get_first_dmapping(), get_next_dmapping()*). Afterwards the virtual machine monitor has to write this list to the buffer reserved within the shared page of the considered virtual machine. Finally it initiates the suspension process by setting the virtual machine state to VM_STOP. Setting the virtual machine's state is done by invoking *vm_set_thread_state(state)* from within the virtual machine scheduler.

Halt and Restart

The second part is done by a component we shortly introduced in section 4.5.3: The inplace-monitor. We extended the inplace-monitor in such a way it can access the compiled list of threads and the aimed virtual machine status. If it recognizes a *VM_STOP* status it processes the current thread list and thereafter stops every virtual machine related thread. Stopping a thread is performed by calling *L4_Stop(threadid)*, a convenient function of the L4 micro-kernel based on the system call *L4_ExchangeRegisters()*.

The counterpart of stopping is restarting the virtual machine. The virtual machine monitor initiates the restarting process by setting the virtual machine state to *VM_START*. The inplace-monitor in the following starts the threads on the current thread list mentioned above by invoking *L4_Start(threadid)*, a further convenient function of the L4 micro-kernel based on the system call *L4_ExchangeRegisters()*. By setting the virtual machine state thus the virtual machine monitor has the requested feature to suspend and restart a virtual machine.

4.5.5 Virtual Machine Migration

Virtual machine migration is closely related to virtual machine suspension. As we demonstrate in this section we implemented virtual machine migration and suspension in such a way they can share some of the required operations. Virtual machine migration is also separated in parts which are done by the virtual machine monitor and parts which are done by the inplace-monitor. For our prototype virtual machine migration is implemented in the function *migrate_vm_vcpu(virtual_machine, vcpu, cpu).*

Initiating Migration

Live migration of virtual machines [6] offers a staged migration approach for virtual machines. As we only migrate threads between local CPUs just some of the stages are of interest for us. Basically virtual machine migration is performed by remapping the virtual CPUs to other physical CPUs than before and afterwards migrate all threads to this new physical CPU.

In stage 0 called pre-migration we identify our target CPU. This stage typically is reserved to be implemented by the virtual machine scheduler. It is the one to determine a new target CPU due to its policy. In stage 1, the reservation, we perform two tasks. At first we reuse the thread list compiler to create the thread list of the virtual machine. However, this list now depends on the CPU a thread is running on. We only migrate the threads of the defined CPU rather than of the complete virtual machine. Thus our interface to read the thread list slightly changed to the function pair *get_first_mapping(cpu), get_next_mapping(cpu).* Furthermore this change requires to monitor the physical CPU a thread is created on. Therefore we can extend the corresponding feature implemented for virtual machine suspension as described above. After we wrote the thread list to the buffer in the shared page we set the virtual machine status to *VM_MIGRATE* , still by invoking *vm_set_thread_state(state).* Before doing so, we additionally have to call *migrate(vcpu, cpu)* which is implemented for each virtual machine. This function sets the new mapping of virtual CPU to physical CPU.

Performing Migration

The second part again is performed by the inplace-monitor of the current virtual machine. This time the virtual machine's state variable is set to the *VM_MIGRATE* state. Now stage 3 - stop-and-copy - begins. This stage is strongly stripped as we have only to stop the threads. The copy part is replaced by the migration of the threads belonging to the virtual machine. We migrate a thread by calling a further convenient function of the L4 micro-kernel called *L4_Set_ProcessorNo(threadid, cpu)* which is based on *L4_Schedule().* This function migrates a thread to the specified CPU. Afterwards stage 4, the commitment, informs the virtual machine monitor about the successful migration process by setting the virtual machine state to

VM_SELF. This indicates the virtual machine monitor the availability of the migration (or suspension) part implemented in the inplace-monitor. Finally stage 5 called activation starts all thread of the virtual machine again and the migration process is completed. The prototype supports migration of multiple virtual CPUs at once as the thread list can be compiled per virtual CPU.

A remark: The selection of threads which have to be migrated in order to receive a running system after a migration process can differ from one virtualization environment to another. The L4Ka virtualization environment requires to migrate all user-level threads related to the considered virtual machine: All application threads, the kernel thread, the irq dispatcher thread and the inplace-monitor itself.

4.6 Host-Level Policies: Throttling and Balancing

We discussed the basic mechanisms available for the virtual machine scheduler to implement scheduling policies: Energy budgets, virtual machine suspension and virtual machine migration. In the following we demonstrate the implementation of two basic scheduling policies for host-level scheduling which we already introduced for guest-level scheduling. Both scheduling policies will be evaluated in Chapter 5.

4.6.1 Energy-Aware Throttling

Energy-aware throttling on host-level offers additional control to the virtual machine scheduler over its guests. It can be utilized if the scheduler faces the issue of non-cooperative virtual machines, for example in case a virtual machine does not support guest-level scheduling or it is infected by a virus. Our implementation is capable to enforce energy budgets for each CPU and each virtual machine.

Implementing a basic energy-aware throttling policy having energy budgets, energy profiles and a suspension mechanism for the considered scheduling principal, the virtual machine, is straight forward. We have a suspension mechanism for virtual machines, energy budgets for virtual machines and their energy profiles. As seen in Figure 4.8 the virtual machine scheduler is called every collector tick. Thereby we receive a low reaction latency. The scheduler itself compares the current energy profiles of the virtual machine to the energy budget of the virtual machine. In case a rule violation is detected for the virtual machine the energy-aware throttling procedure is called. This procedure detects all virtual machines which consume too much energy and initiates suspension afterwards.

Suspended virtual machines have to be restarted again. Otherwise they sleep from the point of suspension. We implemented the restarting logic within the scheduler. It probes at first if there is a virtual machine already suspended. If this is the case it looks if energy is still available and thereafter switches the state of the virtual machine to *VM_START*.

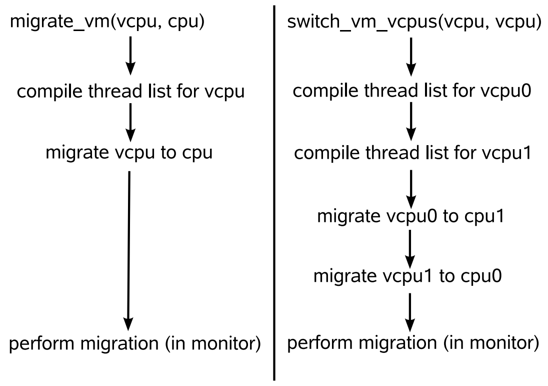
migrate_vm(vcpu, cpu)
compile thread list for vcpu
migrate vcpu to cpu
perform migration (in monitor)
switch_vm_vcpus(vcpu, vcpu)
compile thread list for vcpu0
compile thread list for vcpu1
migrate vcpu0 to cpu1
migrate vcpu1 to cpu0
perform migration (in monitor)

Figure 4.9: Switching the virtual CPUs: If a virtual machine covers all physical CPUs we require switching the virtual CPUs in order to support energy-aware balancing.

4.6.2 Energy-Aware Balancing

Energy-aware scheduling on host-level is required to balance the energy consumption of different physical CPUs. In contrast to this, energy-aware balancing on guest-level only provides balancing the virtual CPUs. This, however, is not enough for the case we have more physical CPUs then the virtual machines cover with their virtual CPUs. In this case we require a mechanism to exhaust energy capacities on other available physical CPUs, too. Our implementation offers such a mechanism, virtual machine migration, and thus it is capable to ensure all physical CPUs consume the same amount of energy.

As discussed for the collector's reuse of energy profiles for virtual machines and for physical CPUs in section 4.5.2 we have the energy consumption of both scheduling entities available in time. The scheduler is again called on every collector tick. Thus we again receive a low reaction latency. In a first step the scheduler tests if there is a rule violation. In case there is a rule violation, the scheduler calls the virtual machine migration routine *migrate_vm_vcpu(virtual_machine, vcpu, cpu)*. This routine searches for a physical CPU which has energy capacities left. In the following it migrates virtual machines which have energy capacities left to this found CPU. For our experimental environment in Chapter 5 we additionally implemented an extended version of *migrate_vm_vcpu(virtual_machine, vcpu, cpu)* in order to migrate a virtual machine called *switch_vm_vcpus(virtual_machine, vcpu, vcpu)*. This routine is capable to switch two virtual-to-physical-CPUs mappings at once. It allows to demonstrate the practicability of virtual machine migration also in the case each virtual machines' virtual CPUs cover all physical CPUs. As Figure 4.9 shows the proceeding however remains quite similar to *migrate_vm_vcpu(virtual_ machine, vcpu, cpu)*. After we compiled the thread list now for *both* virtual CPUs we call the migration routine of the virtual machine which afterwards initiates the migration process in the inplace-monitor (see section 4.5.5).

Besides the requirement for our experimental environment this solution is necessary to be able to migrate multiple virtual CPUs to another physical CPU in one single scheduling decision. Dividing this problem for n virtual CPUs into n migration processes is not possible as we get another result then. The threads of first the migrated virtual CPU already would change the energy consumption on the second virtual CPU eventually. Thus the scheduler could come to another decision as within the last scheduling period. Let us consider the following example using *migrate_vm_vcpu(virtual_ machine, vcpu, cpu)*. Virtual CPU 0 is mapped to physical CPU 0 and virtual CPU 1 is mapped to physical CPU 1. If virtual CPU 0 was migrated to physical CPU 1 in scheduling period 1, the virtual machine scheduler consequently detects a higher energy consumption on physical CPU 1 in the next scheduling period. The threads of virtual CPU 0 are now running on physical CPU 1. Thus instead of migrating virtual CPU 1 to physical CPU 0 it decides to migrate both virtual CPUs to physical CPU 0. This result is a different mapping of virtual CPUs to physical CPUs then the expect one if we use *switch_vm_vcpus(virtual_machine, vcpu, vcpu)*: Virtual CPU 1 should be running on physical CPU 0 and virtual CPU 0 should be running on physical CPU 1.

Chapter 5

Experimental Results

The goal of our work, to provide a framework, which supports *energy-aware scheduling for virtual machines in multiprocessor environments* is evaluated in this chapter. We conducted experiments to demonstrate our approach achieves this goal. We consider different scenarios for the guest-level scheduling policies and the host-level scheduling policies we discussed in the implementation chapter. Furthermore we assess the energy-overhead our implemented prototype induces to our system. Additionally the performance of our base mechanism on host-level (virtual machine suspension and virtual machine migration) is evaluated. Finally we give an analysis of the advantages of guest-level scheduling vs. host-level scheduling. This chapter presents our experiments and discusses the results we found.

5.1 Experimental Environment

The experimental environment is the same we used for the implementation of our prototype. We use the L4Ka virtualization technology which is built upon the fast *L4Ka::Pistachio micro-kernel* [20]. We can benefit from the stability of this system as well as from its proved rapidness of its base mechanism (such as IPCs). The well elaborated virtualization environment provides additional features like device driver reuse which simplifies the controlling of the measurements in different cases. The guest operating systems are L4Linux instances which are para-virtualized in a way to work with our virtualization architecture. It bases upon a Linux 2.6 kernel. To get an completely bootable system available we use a Debian installation. The hardware consists of an Intel Pentium D system, clocked with 3 GHz on both CPUs. Furthermore the system is equipped with 2 GByte memory. We measured a maximum power consumption of 58 watt for each CPU.

Our experiments often execute an application called bitcnts. Bitcnts is a simple application which creates CPU load by making extensive use of bit shifting functions of the CPU. Thereby it creates CPU load and consumes around 52 watt. All experiments are measured in intervals of 5000 msecs until further notice.

We conducted the experiments in such a way that we control the energy con-

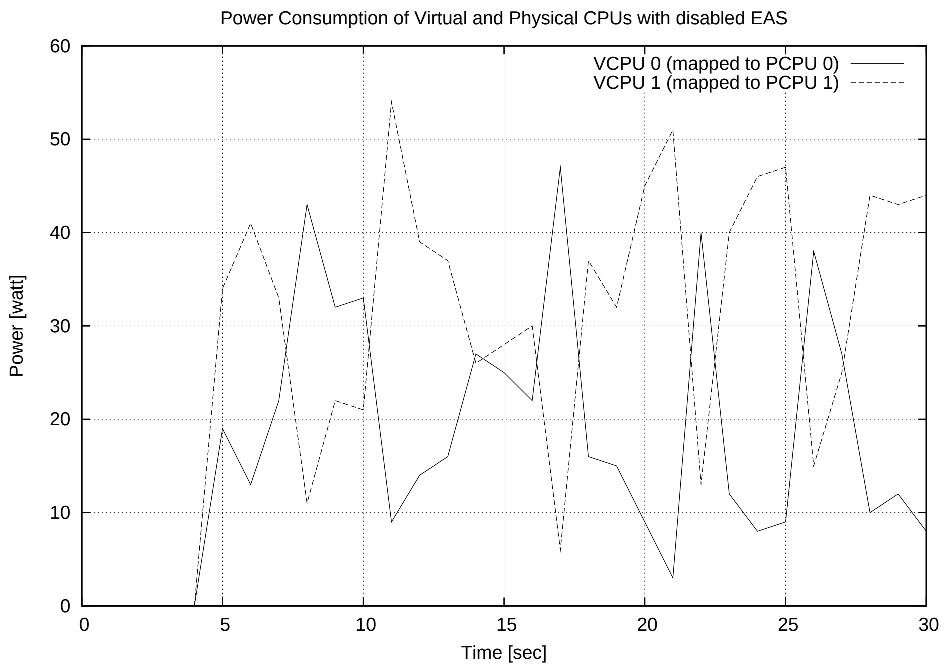

Figure 5.1: Starting point: The power consumption of our experimental environment if energy-aware scheduling is disabled. There is no balancing of the energy consumption at all, neither on the virtual CPUs nor consequently on the physical CPUs (both virtual CPUs are mapped idempotent to the physical CPUs).

sumption for the discussed scheduling entities. We assign energy budgets which results into a power consumption of the same value (for example if we assign an energy budget of 25 joule this results into an average power consumption of 25 watt). Power consumption is a more meaningful value and can be better evaluated. Therefore we monitor the power consumption for these scheduling entities rather than the energy consumption.

To visualize the starting point for our evaluation we measured the behavior of the power consumption for our experimental environment occurring in case the new scheduling mechanisms are disabled. To conduct the experiment we started 10 instances of bitcnts and monitored the power consumption of the virtual CPUs and the one of the physical CPUs. But as Figure 5.1 shows there is no energy-aware behavior at all. In fact it is worse. If you take the average of both virtual CPUs, virtual CPU 0 has a significant higher power consumption than virtual CPU 1. This is due to the natural behavior of the Linux scheduler which uses a performance centric heuristic based upon the CPU load. The CPU load however, is found by analyzing the length of the CPU's runqueue. Thus we conclude there is no relation between the migration criterion of the Linux standard scheduler (CPU load) to the one we use for energy-aware scheduling: The energy consumption.

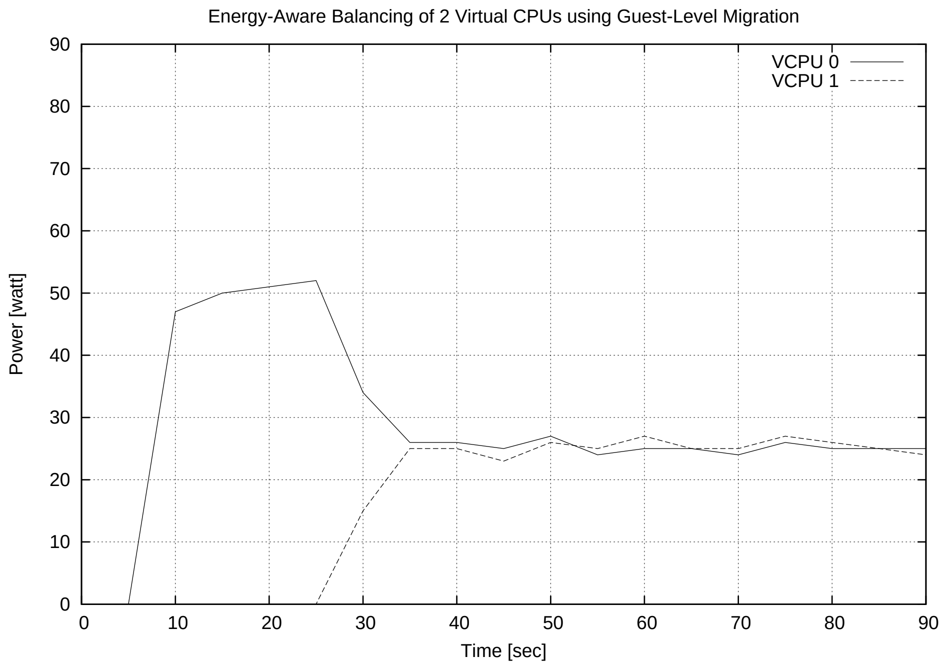

Figure 5.2: Balancing energy consumption of two virtual CPUs: After a period of free scheduling (30 secs) the guest scheduler balances both virtual CPUs

5.2 Guest-Level Scheduling

This section evaluates the prototype's implementation of guest-scheduling. We implemented two basic scheduling policies which prove that energy-aware scheduling policies can be implemented upon our framework. In the following we demonstrate its correct behavior for different scenarios. We start the evaluation of the balancing policy followed by the throttling policy. We finalize this section by proofing that energy-aware balancing is capable to exhaust existing energy capacities of virtual CPUs.

5.2.1 Functionality: Balancing Energy Consumption

Energy balancing of virtual CPUs is to keep their energy consumption even. To show this we assigned a virtual machine 2 virtual CPUs. After a successful boot process we start an instance of bitcnts. Initially energy-aware scheduling is deactivated. The process is running only on virtual CPU 0. Thus only virtual CPU 0 is consuming a considerable amount of energy. After a period of 30 seconds guest-level scheduling is activated. The energy balancing algorithm begins to work and the thread is assigned energy-aware to both virtual CPUs by the guest's scheduler. As Figure 5.2 illustrates the application initially consumes about 50 watt upon virtual CPU 0. 30 seconds later the scheduler begins to distribute it in a way both virtual CPUs consume around 25 watt. The deviance between the energy consumption of both virtual CPUs is between 0 and 1 watt. This deviance is related to the scheduling interval. Currently we are migrating threads to virtual CPUs around every 10 msecs. Thus a power overhead up to 1 watt upon each virtual CPU is reasonable.

5.2.2 Functionality: Throttling Energy Consumption

Another mechanism we analyze in the following is throttling within guests. The settings for this scenario are a little different from the one before. We do not only balance the power consumption of both virtual CPUs. Additionally we limit their power consumption by using guest-level suspension. The energy budget of Virtual CPU 0 is limited to 15 joule and virtual CPU 1 to 25 joule demonstrated in Figure 5.3. Again we start an instance of bitcnts. It consumes energy unthrottled on virtual CPU 0 until after 30 seconds guest-level scheduling is activated. Now the thread is migrated such that energy is consumed on both CPUs. However, the virtual machine's energy consumption on virtual CPU 0 is limited. Therefore it only has a power consumption up to 15 watt. Consequently virtual CPU 1 should consume around 35 watt. But this virtual CPU's power consumption is limited to 25 watt. Thus energy-aware throttling by the guest itself is behaving as expected.

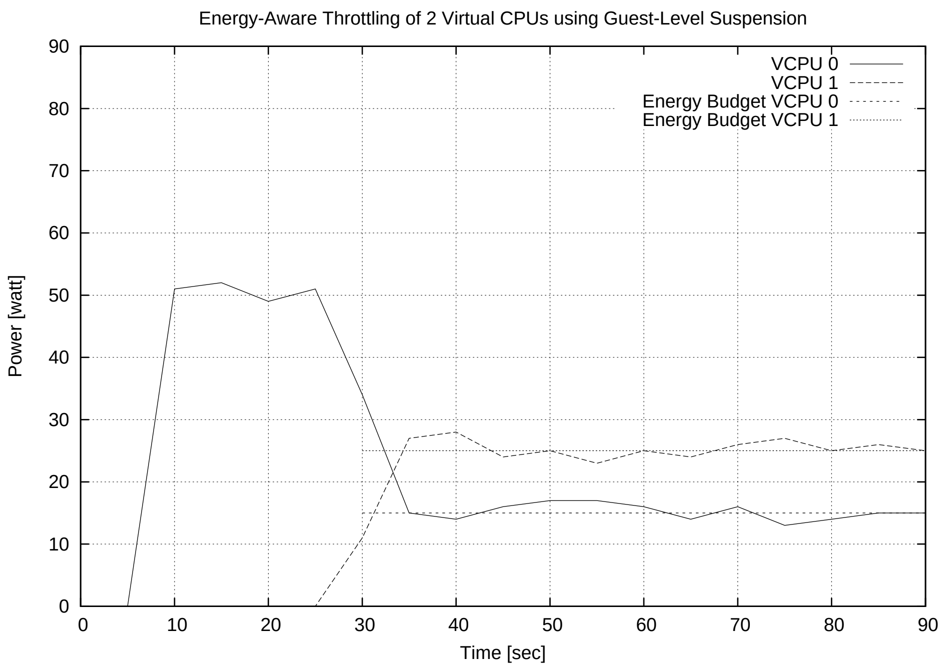

Figure 5.3: Throttling energy consumption of two virtual CPUs: At first two virtual CPUs are consuming energy unthrottled. After 30 seconds virtual CPU 0 is throttled to an energy budget of 15 joule and virtual CPU 1 is throttled to 25 joule.

5.2.3 Exhausting Virtual CPU Capacities

The last both experiments showed that the basic functionality of guest-level scheduling is available and works as expected. But, in order to show we can participate from the higher computing power of multiprocessor systems, we need to demonstrate in the following, the guest-level scheduler is capable to exhaust the offered energy capacities of such a system.

The experiment is similar conducted as those before. However, this time we run 10 instances of bitcnts. There is no budget defined. This experiment should result into power consumption of around 50 watt on both CPUs. As Figure 5.4 shows this is the case. The guest-level scheduler migrates the threads in a way both virtual CPUs are completely exhausted.

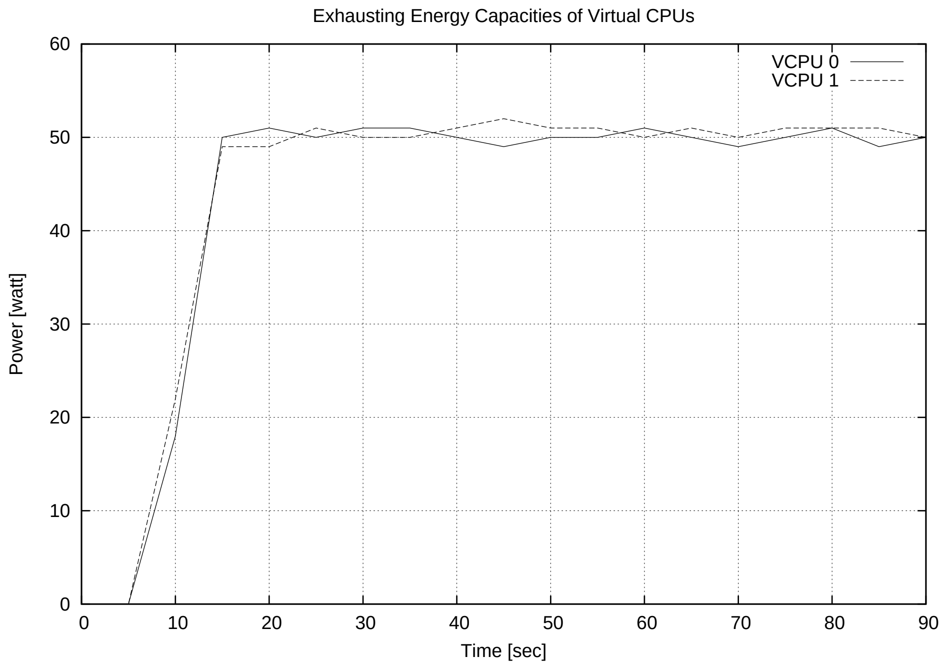

Figure 5.4: Exhausting virtual CPU capacities: After starting 10 applications the energy-aware guest-level balancer begins to distribute them to virtual CPUs so they exhaust the capacities of both virtual CPUs.

5.3 Host-Level Scheduling

Host-level scheduling is the complementary part to guest-level scheduling used in case a guest can not schedule energy-aware. Enforcement of energy budgets and applying global informations (such as a physical CPU's temperature) for scheduling purposes moreover only can be solved using host-level scheduling.

In this section we demonstrate that energy-aware scheduling policies can be implemented if they rely upon our framework. Therefore we implemented energy-aware balancing and energy-aware throttling for our prototype which is adducted for evaluation now. To evaluate an important feature of host-level scheduling, the enforcement of guest energy budgets by the virtual machine scheduler, we furthermore conduct an experiment running an energy-aware virtual machine and a non-energy-aware virtual machine.

5.3.1 Functionality: Balancing Energy Consumption

The energy-aware balancing policy we implemented for our prototype evens out the energy consumption on multiple physical CPUs. Our balancing policy thereby relies upon virtual machine migration and energy budgets for physical CPUs. In order to keep the results of guest-level scheduling and host-level scheduling comparable we tried to reuse the guest-level scenarios as far as possible.

After a finished boot process an instance of bitcnts is started. Initially host-level scheduling is deactivated. Therefore the power consumption on physical CPU 0 is around 50 watt and the one on physical CPU 1 is 0 watt. After a time period of 30 seconds the host-level scheduler is activated and begins to even out the energy consumption of both physical CPUs by migrating the virtual machine. Figure 5.5 shows that the power consumption on both physical CPUs is balanced to around 25 watt. However, in contrary to using guest-level scheduling the deviance now is between 0 and 4 watt. This can be reasoned by the higher scheduling interval which we set around a quarter second for host-level scheduling.

5.3.2 Functionality: Throttling Energy Consumption

Energy-aware throttling can be used to ensure a guest can not consume more (or better: Significantly more) energy than its assigned energy budget. For the implementation we utilized virtual machine suspension and energy budgets for virtual machines.

We once again set up the same scenario as for guest scheduling as far as possible. Virtual machine 0 is enforced to an energy budget of 15 joule while virtual machine 1 may consume up to 25 joule. After the boot process is finished the bitcnts process is started within both virtual machines. Initially the virtual machine scheduler is deactivated. Both virtual machines can consume energy freely. As Figure 5.6 shows after 30 seconds energy-aware throttling is activated. In the following virtual machine 0 consumes around 15 watt and virtual machine 1 around 25 watt. For the deviance the same holds as for energy-aware balancing. The backing mechanism is the same and operates again with a scheduling period of 250 ms. However, enforcement of energy budgets is possible if the guest does not behave energy-ware. We additionally demonstrate this in more detail in the next section where guest-level mechanism and host-level mechanism are used in parallel.

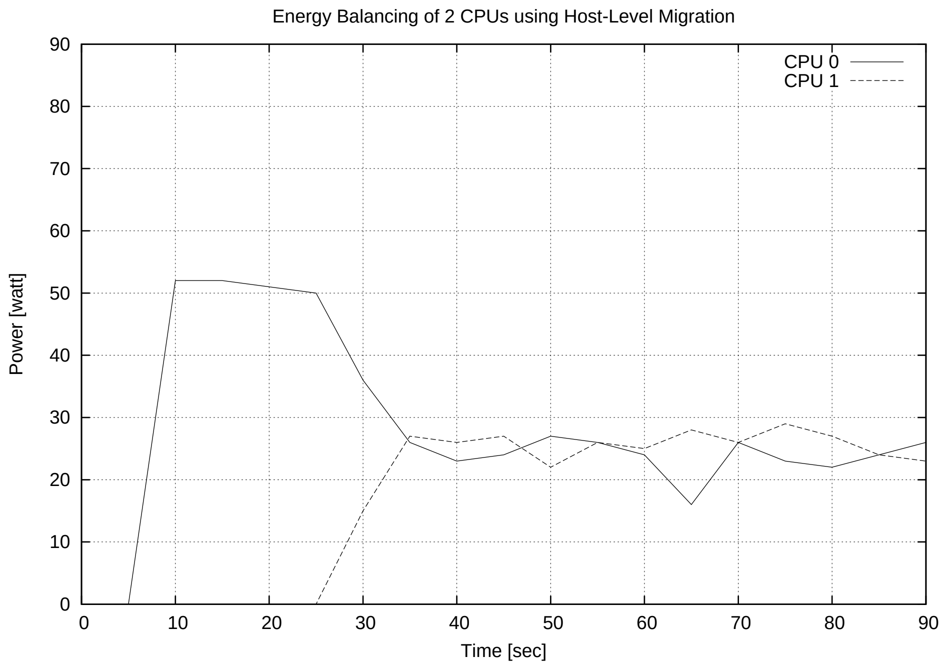

Figure 5.5: Balancing energy consumption: After a period of 30 seconds the host-level scheduler tries to balance the power consumption of two physical CPUs.

5.3.3 Throttling on Host-Level and on Guest-Level

An important issue for energy-aware scheduling of virtual machines is the control of non-cooperative virtual machines. The lack of a guest-level mechanism solving this issue leads to energy imbalances for virtual machines which are behaving correctly. This results from fact that the non-energy-aware virtual machines consume eventually too much energy. In the following we evaluate the possibility, our prototype offers to overcome such an imbalance using host-level features.

We start by setting up two virtual machines running bitcnts. After creating some load after 10 seconds guest-level throttling begins to keep the assigned energy budgets. But, as Figure 5.7 illustrates after 20 seconds only virtual machine 0 is behaving correctly. It throttles itself to consume only 20 watt. Virtual machine 1 however is running without throttling itself to its assigned budget of 30 joule. Thus it is consuming more energy than its assigned budget and virtual machine 0 receives less computing power as it is entitled to. This can be seen within the time interval 10 to 30 seconds. The virtual machine behaving correctly consequently would be penalized for its correct behavior.

To solve this issue our prototype supports host-level throttling. Host-level throttling can be used to throttle down a virtual machine's energy consumption to a certain limit. After a period of 30 seconds this mechanism is activated. As Figure 5.7 shows the power consumption of virtual machine 1 is throttled to 30

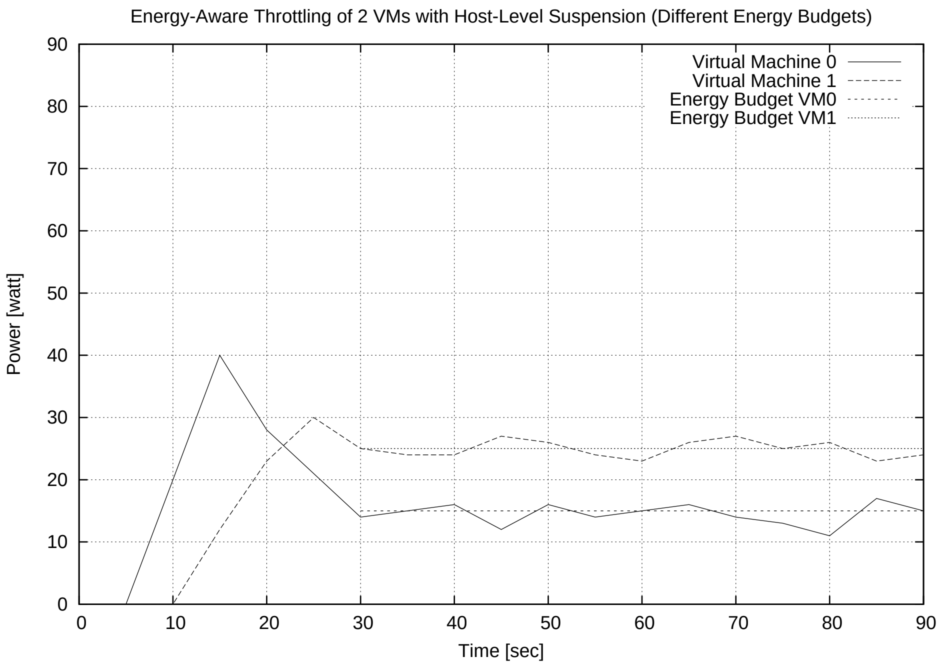

Figure 5.6: Throttling energy consumption: Two virtual machine consume energy unthrottled for 30 seconds. Afterwards host-level suspension regulates VM0 to 15 watt and VM1 to 25 watt.

watt. Thereby virtual machine 0 in the following is no more penalized. It can consume its assigned energy budget. Thus energy-aware throttling on host-level can help to enforce energy budgets for virtual machines in case guest-level scheduling is not available.

5.3.4 Exhausting Physical CPU Capacities

To demonstrate the possibility to exhaust the energy capacities of physical CPUs we booted 4 virtual machines running 5 instances of bitcnts each. After a period of 20 seconds all virtual machines are up and the bitcnt instances are running. There are no budgets assigned to the virtual machines. They all may consume as much as they can get. This experiment demonstrates the energy capacities on both physical CPUs are exhausted and even more they are balanced. This can be seen beginning from time period 20 to 90 in Figure 5.8. Both physical CPUs consume the same power which is around 50 watt each. Thus by using virtual machine migration we are capable to exhaust the capacities of available physical CPUs.

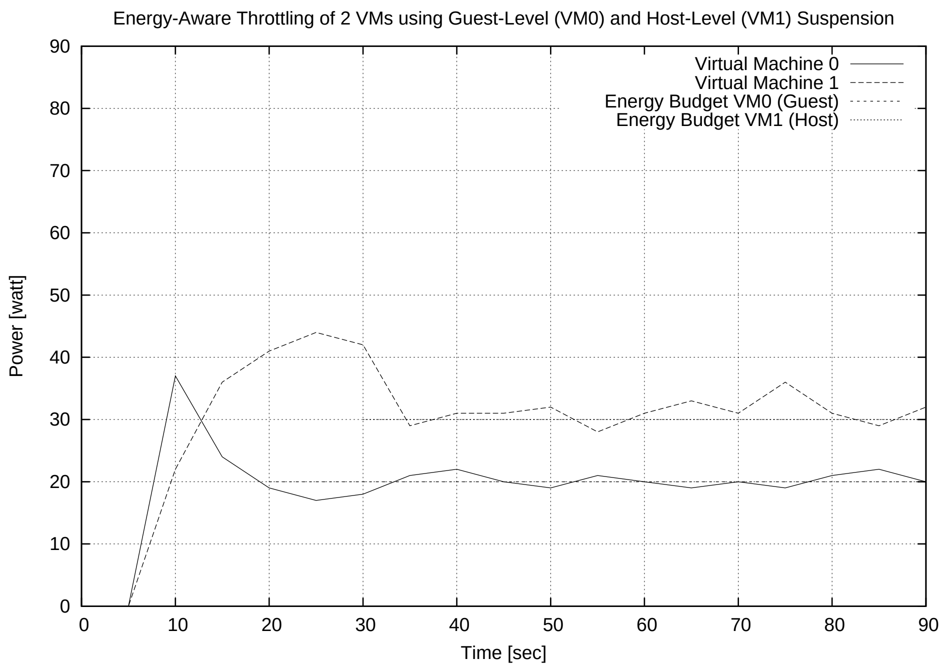

Figure 5.7: Throttling on host-level and on guest-level: 2 VMs are running, however VM1 does not support energy-aware scheduling. After 20 seconds VM0 regulates itself to 20 watt. VM1 still consumes as much energy as possible which punishes VM0. Therefore host-level throttling interferes and throttles it down to 30 watt, which allows VM0 to run as expected.

5.4 Energy Overhead

An important goal we requested in section 3.1 is efficiency of our design. We did not want to induce much energy consumption to our system. Of course a prototype is rarely the best alternative to demonstrate a feature like efficiency. However, we show that even our prototype only induces a low overhead of energy consumption which fortifies our statement. To demonstrate this we evaluated the energy consumption induced by the virtual machine in case we use guest-level migration or host-level migration.

5.4.1 Energy Overhead on Guest-Level-Scheduling

We begin by evaluating the energy overhead of guest-level scheduling. The experiment is conducted to start bitcnts instances after 10 seconds. After 20 seconds the guest-level scheduler enables energy-aware balancing. The scheduler tries to even out the energy consumption of the virtual CPUs. We evaluated this issue for 1, 2 and 4 virtual machines to see if there is a significant increase due to the number of virtual machines.

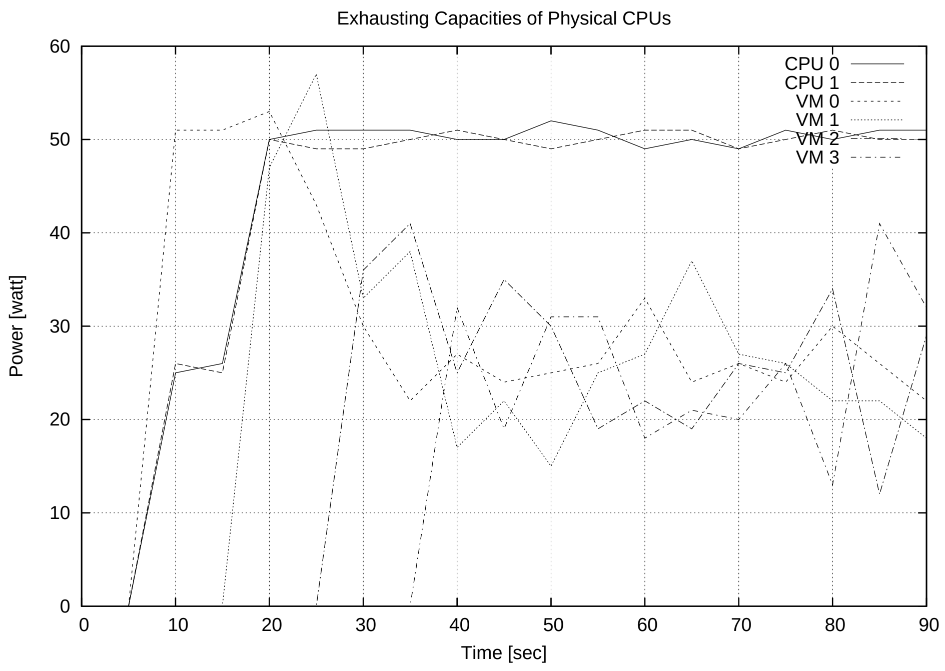

Figure 5.8: Exhausting physical CPU capacities: After starting 4 virtual machines running five bitcnts instances each the energy-aware host-level scheduler begins to distribute them to physical CPUs so they exhaust the capacities of both physical CPUs. Remark: We did not(!) balance the virtual machines' energy consumption in this experiment!

At first we relate to the measurement for one virtual machine running. Regarding time interval 0-10 seconds, Figure 5.9 shows the power consumption if the virtual machines are idle. The power consumption is around 0.8 watt. After 10 seconds bitcnts instances are started. This is expressed by a low increasing to 0.85 watt. At 20 seconds the guest-level scheduler is activated and thus the power consumption increases to around 1 watt. This is the maximum we ever measured. If we take in consideration, that there is only an increase of 0.2 watt from having the virtual machines in idle state to the performing of energy-aware scheduling, we argue this is negligible.

The second fact which is really considerable is the reduction of the power consumption of the virtual machine monitor if we increase the number of virtual machines. Currently we do not have an explanation why this is happening. However, as the overhead is going down rather than it increases, we can state the overhead induced by the virtual machine monitor is around or below 1 watt.

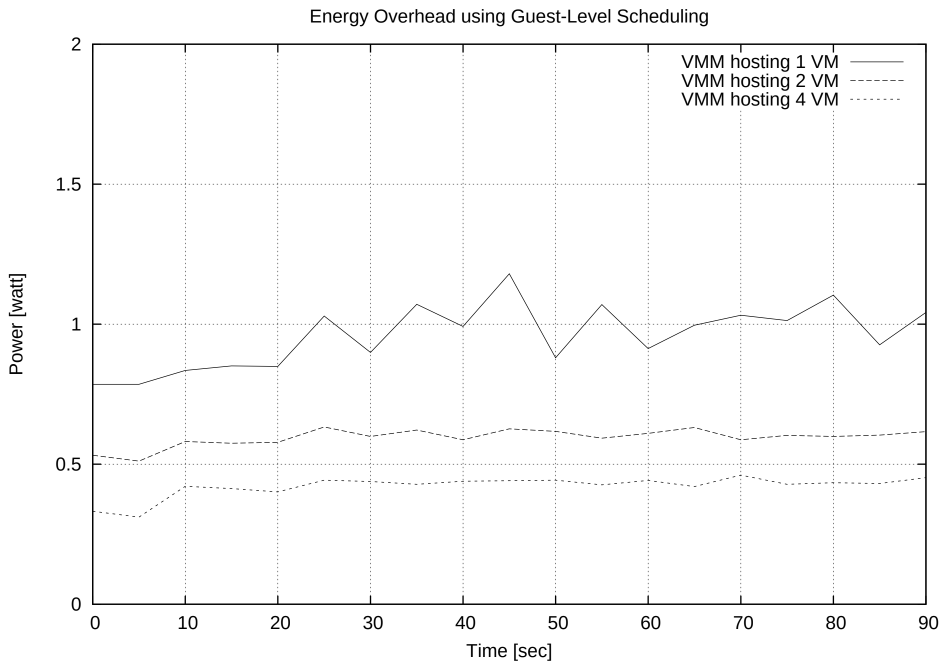

Figure 5.9: Virtual machine monitor's energy overhead using guest-level scheduling: The virtual machine monitor does not consume significantly more power because of the appliance of guest scheduling after 20 seconds.

5.4.2 Energy Overhead on Host-Level-Scheduling

The next point of interest is the increase of the energy overhead when using host-level scheduling induced by the virtual machine monitor. We take into consideration the energy consumption of one and two virtual machines. The other settings are the same as for guest-level scheduling in the section below.

The power consumption of the virtual machine monitor is around 0.8 watt [see time interval 0-10 in Figure 5.10]. This is a little more compared to guest-level scheduling. The virtual machine monitor needs to perform extra controlling of energy budgets here (for example comparing the consumed energy of virtual machine to its assigned energy budget). After ten seconds the bitcnts instances are started. This results into some additional, yet low, overhead. It is even a little more if two virtual machines are started. This peak is due to the monitoring we do for the compilation of the thread list [see section 4.5.4 and 4.5.5]. After 10 more seconds the virtual machine scheduler limits the energy consumption of each virtual machines. However the activation of the host-level scheduler does not induce any significant further energy consumption to our system. This is due to the fact, that there is nothing further to do then to compile the thread list. This is documented in Figure 5.10, interval 20 to 90 seconds, as the power consumption does not increase.

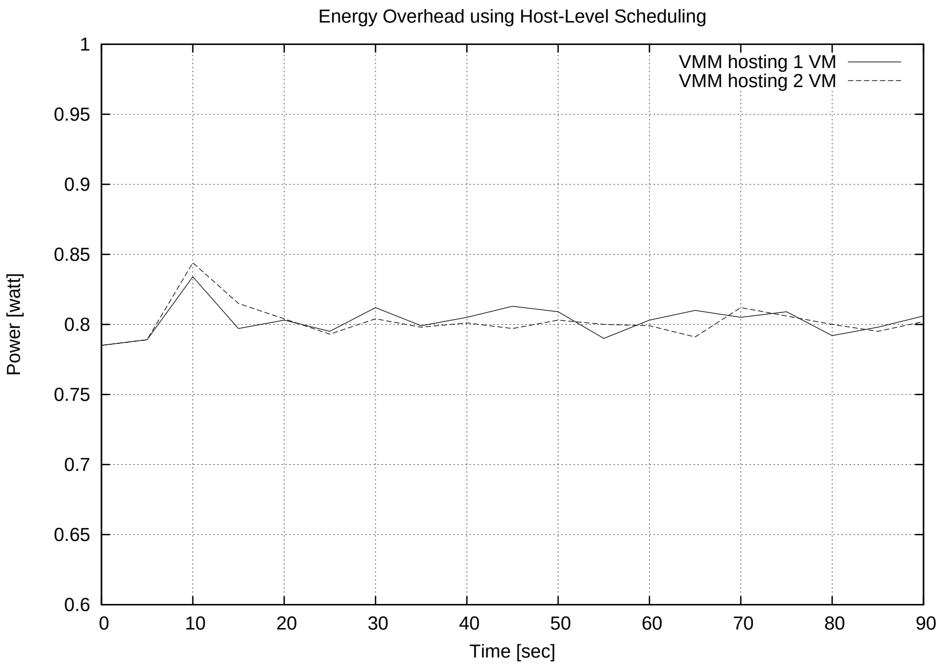

Figure 5.10: Virtual machine monitor's energy overhead using host-level scheduling: The virtual machine monitor continues to consume around 0.8 watt if the host-level scheduling is applied after 20 seconds.

5.5 Performance

In this section we evaluate the performance of our prototype's virtual machine scheduler. We conducted two experiments: One for virtual machine migration and one for virtual machine suspension. We did so as we found out that they behave in different ways as we demonstrate below. These two measurements have quite an impact if one implements new scheduling policies. They demonstrate the minimal reaction time which has to be considered if one decides which part of a scheduling policy has to be implemented within the guest-level scheduler and which part within the host-level scheduler.

5.5.1 Performance of Virtual Machine Migration

For this experiment we started a virtual machine with a diversified number of threads running. We tried to find a relation between the time virtual machine migration requires to migrate a complete virtual machine and the number of the virtual machine's threads. This gives us a hint about the minimal scheduling period if one utilizes virtual machine migration for an energy-aware scheduler.

As Figure 5.11 shows the time required to migrate a virtual machine increases nearly linear with the number of threads. This offers a good prediction model for

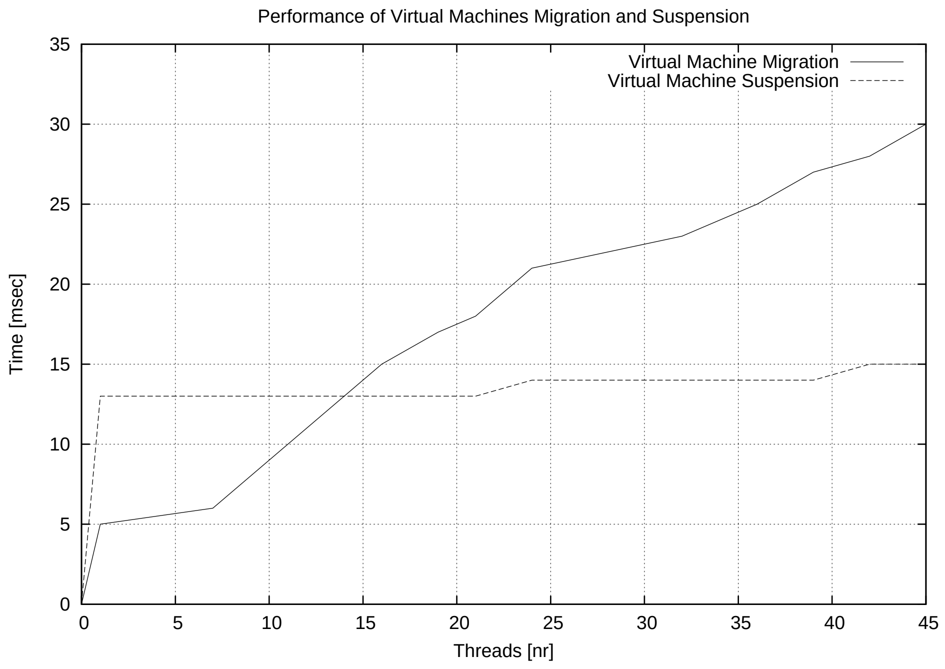

Figure 5.11: Performance of virtual machine migration and suspension: The time required for migrating a complete virtual machine nearly linear increases with number of threads. The time to suspend a complete virtual machine however is nearly constant (with a slow increasing dependent on the number of threads).

the design phase of a new scheduler. It can be deduced a lower boundary for the scheduling period required for host-level migration if the maximum of threads of the system can be estimated. We can predict that virtual machine migration requires 1ms per thread of a virtual machine.

5.5.2 Performance of Virtual Machine Suspension

This experiment is conducted the same as for virtual machine migration. However, now we analyze what happens for virtual machine suspension. These timing values are interesting in order to estimate the preciseness regarding the enforcement of energy budgets which an energy-aware scheduler relying on host-level suspension can grant.

Figure 5.11 demonstrates a result which can be considered quite welcome. The time period required to suspend a virtual machine is nearly constant. It is increasing very slowly with the number of threads. Thus invoking virtual machine suspension can performed with a high frequency. Yet it has to be taken in consideration that a suspended virtual machine needs to be restarted again. Thus the timing has to be doubled at least.

5.6 Host-Level vs. Guest-Level Scheduling

An important point of our design has been the availability of scheduling mechanisms on guest-level and on host-level. We argued we require host-level scheduling (besides others) if a guest is not scheduling energy-aware. We furthermore argued that guest-level scheduling is more accurate and thus is required for precise scheduling decisions. In the following we demonstrate the limits of guest-level scheduling and host-level scheduling in a way to make an estimation, which mechanisms to use, is getting a more transparent task.

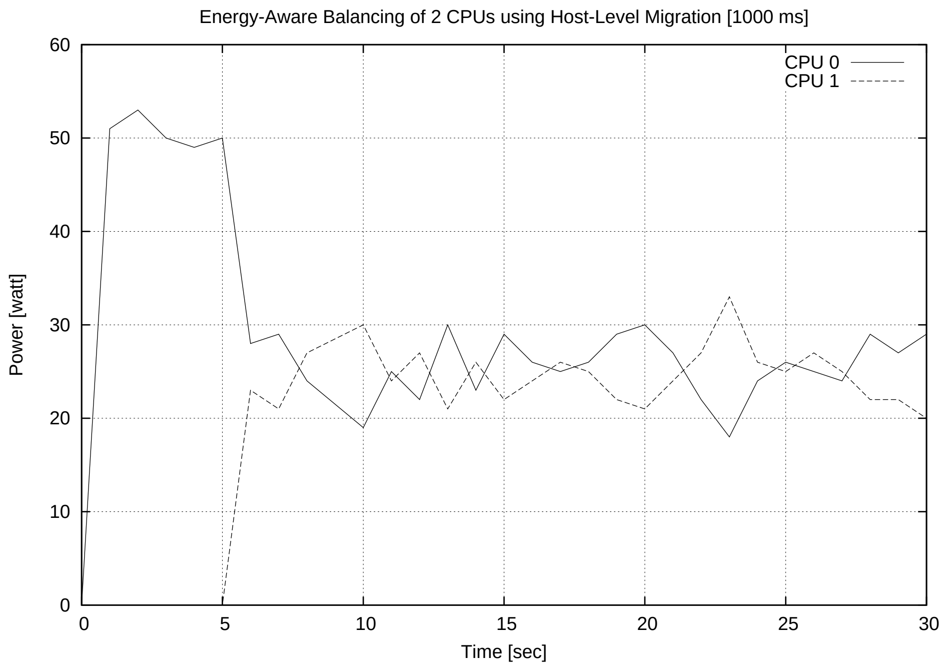

Figure 5.12: Limits of host-level scheduling: Reducing the measurement interval to 1000 ms leads to a significant deviance between the assigned energy budgets and the kept one.

5.6.1 Limits of Host-and Guest-Level Scheduling

At first we conducted an experiment which clarifies the limits of host-level scheduling. The limits of host-level scheduling we found in the granularity of the scheduling decisions. Migrating a complete virtual machine due to its energy budget using host-level scheduling is by far not this exact as guest-level migration.

This experiment basically is related to the one we discussed in section 5.3.1. We demonstrated that our energy-aware scheduler is capable to balance the energy consumption of physical CPUs. This experiment has been performed by using the average sampled for 5000 msecs. In this section we reduce the sampling interval

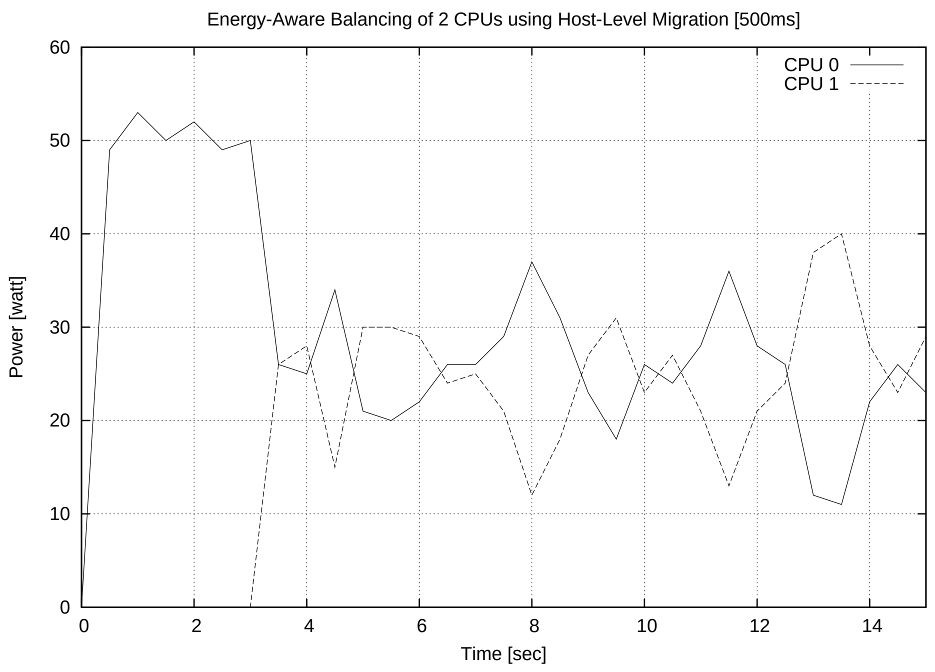

Figure 5.13: Limits of host-level scheduling: Reducing the measurement interval to 500 ms leads to an even more significant deviance between assigned energy budget and the kept one.

to 1000 msecs and afterwards to 500 msecs in order to show the deviance between the energy budget and the kept energy budget increases.

Figure 5.5 documents the energy balancing with a measurement interval of 5000 msecs. As Figure 5.12 illustrates we gain an increasing of the deviance if we reduce the measurement interval. The highest deviance is around 7 watt. Compared to 3 watt for 5000 msecs this is a significant increasing. If we even more reduce the interval to half a second the deviance once more increases to around 11 watt (see Figure 5.13). The explanation is found in the migration interval of virtual machine migration which is currently set to around a quarter second. Therefore only every 250 msecs the virtual machine scheduler can try to even out the power consumption on both CPUs. It is not possible to balance the power consumption of the CPUs having only 2 or 4 scheduling decision. Reducing this scheduling period however is not the solution to this issue. We showed that the required time to migrate threads is increasing with the number of threads running in a virtual machine. A virtual machine with 200 threads already requires around 200 msecs to migrate completely. Thus we have to conclude virtual machine migration is a coarse-granular mechanism to even out the power consumption of CPUs.

The limitations of guest-level scheduling are rather found in the functionality itself than by the impreciseness of the mechanism. As Figure 5.14 demonstrates guest-level scheduling keeps balancing the power consumption of its virtual CPUs

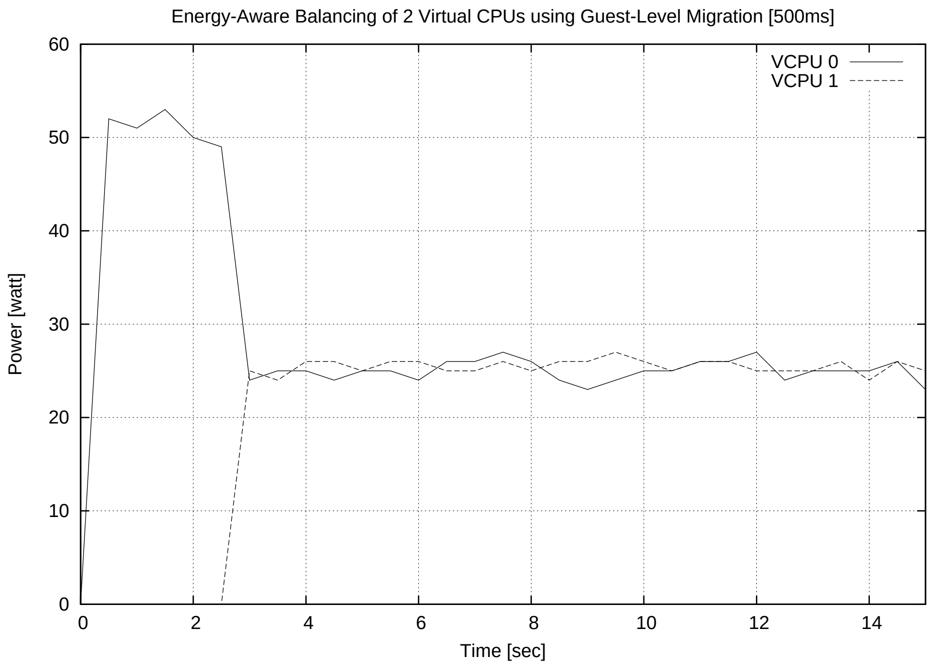

Figure 5.14: Limits of guest-level scheduling: Reducing the measurement interval to 500 ms still offers the requested balancing. The limits of guest-level scheduling we found in the functionality of the mechanism, not in its impreciseness.

even if we reduce the measurement interval down to half a second. This can be reasoned by the fact hat scheduling decisions for virtual CPUs can be made with a much higher frequency. Currently, we use a scheduling interval of 10 msecs for guest-level scheduling. Thus every 10 msecs we come to a decision if threads have to be migrated to a potential less energy consuming virtual CPU. Additionally we do not require to migrate all threads, as we have to for virtual machine migration. The Linux scheduler only migrates some of them in order to balance the power consumption. This reduces the time required for guest-level balancing again.

However, guest-level scheduling is not capable to even out the power consumption of physical CPUs generally. Only if there is an idempotent mapping of virtual CPUs and physical CPUs (that is virtual CPU 0 is mapped to physical CPU 0, virtual CPU 1 to physical CPU 1, and so on) and furthermore just one virtual machine is executing in parallel, this goal can be achieved. Running only 1 virtual machine stands in contrast to the definition of virtual machines. The purpose of virtual machine technology is to execute multiple operating systems in parallel. Therefore the limitation of guest-level scheduling is rather due to its functionality than due to its capabilities.

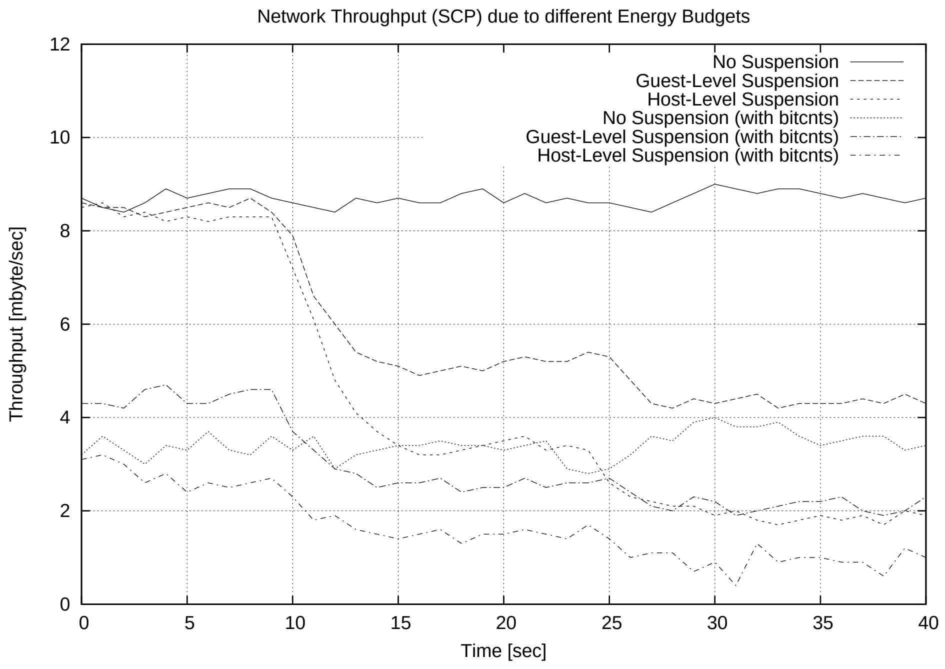

Figure 5.15: Network throughput: If two different applications, one consuming much energy (bitcnts) and one consuming less energy (scp), are competing for computing power the assignment of host-level scheduling is unfair. Guest-level scheduling however assigns both the same amount computing power. This fact is represented by the network throughput of the scp instance. The throughput has been measured for an average power consumption of 25 watt (no limitation), 15 watt and 10 watt.

5.6.2 Application Benchmark

The previous experiment revealed the difficulties of host-level scheduling to equally distribute computing power if we consider small measuring intervals. We found a further issue of host-level scheduling, when we considered the differences between both scheduling levels concerning the assignment of computing power to threads. Once again the more coarse-grained approach of host-level scheduling, which is only controlling virtual machines, has significant disadvantages. For the next experiments we use secure copy (scp) in order to transfer files to the virtual machines. Using our virtualized L4Linux we gain a network throughput of around 9 mbytes/sec if all our mechanisms are disabled. The virtual machine then consumes nearly 25 Watt.

Network Throughput

To visualize the assignment of computing power by the different scheduling-levels we arrange an experiment copying a big file to a virtual machine using scp. We monitored the network adapter's throughput and assigned different energy budgets to the virtual machine to find if there are differences between both scheduling levels.

In the time interval 0-10 we did not assign any budget. As Figure 5.15 illustrates the network throughput is the same for guest- and host-level scheduling disabled as well as if one of them is enabled. It always is around 8.7 mbytes/sec. The virtual machine consumes in this period around 25 watt. Beginning with second 10 we throttled it to 15 watt. The virtual machine then has a throughput of around 5.2 mbytes/sec for guest-level scheduling and around 3.2 mbytes/sec for host-level scheduling. The values diverged even more when we throttled it down to 10 watt. For guest-level scheduling the consumption is reduced to around 4.3 mbytes/sec and for host-level scheduling to 2.0 mbytes/sec.

We reason the differences of the network throughput by implementation issues. As we have to halt and restart all threads of a virtual machine using host-level scheduling rather than only a single thread when using guest-level scheduling, a bigger performance loss is induced by the scheduling process. This performance loss is a quite significant value as can be seen in Figure 5.11. Thus the loss of performance results into the loss network packets.

Network Throughput using different Applications

To further support our analysis we considered the network throughput when a second application, which consumes much energy (bitcnts once more), is running in parallel to the scp instance. The budget assignments are the same as for the measurements without bitcnts. The difference of the network throughput with bitcnts as additional energy consumer seen in Figure 5.15 is due to the following issue: Guest-level suspension and host-level suspension result into different assignments of computing power to the threads. To further explain this point we consider an example.

We have an application A (bitcnts) which is consuming 50 watt in case it is allowed to run unthrottled. Furthermore we have an application B (scp) consuming significantly less: 20 watt. Additionally, we set a budget of 20 watt. In case we use guest-level scheduling to keep this budgets, the guest-level scheduler ensures both applications consume 10 watt. Thus the scp instance (application B) transfers only half of the packets than before (for the relation between energy consumption and network throughput of scp please additionally refer to [16]).

If we use host-level scheduling to enforce the energy budget of the virtual machine the guest-level scheduler does not know about this fact. It schedules the threads with the same ratio as before without energy budgets: With a ratio of 2 to 5. Thus application B consumes nearly 5-6 watt. Consequently scp can only transfer a little more than a quarter of its normal throughput. As Figure 5.15 shows host-level scheduled guests always have a lower (close to a half) network throughput if bitcnts is running in comparison to guest-level scheduled guests. This is what we can expect when we transfer the power ratio linearly to the network throughput.

Network Throughput using the same Applications

To fortify our analysis we evaluate a scenario running two instances of scp. We ran a virtual machine: One time using guest-level scheduling and one time using host-level scheduling once more. At first, we analyzed guest-level scheduling. As Figure 5.16 illustrates the throughput for scp is sinking when we reduce the energy budgets. However, the energy remains evenly shared: The power ratio is 1:1 as both applications are scp instances. Afterwards we measured the same experiment using host-level scheduling. As we have two applications of the same kind the guest-level scheduler assigns them both the same amount of computing power. Now host-level scheduling achieves the same results like guest-level scheduling. Both scp instances have the same network throughput for guest-level scheduling and host-level scheduling.

This result fortifies our statement that the host-level scheduler keeps the assignment of computing power of the guest-level scheduler. This time, both threads consume the same amount of energy, as they are instances of the same applications. Therefore the network throughput only is reduced to the half for both scp instances. Due to these facts we argue, a righteous distribution of computing power to threads can only be achieved using guest-level scheduling.

Conclusion

Due to the perceptions of the last paragraph we state it is preferable to implement scheduling policies in the guest's scheduler. At first, the scheduling policies can benefit from the smaller scheduling periods (see section 5.6.1). Furthermore, the distribution of computing power is more righteous (see section 5.6.2) resulting into a better performance for lower energy consuming applications.

However, the replacement of host-level functionality by guest-level functionality only is applicable if the host-level scheduler supports this procedre. For example if the host-level scheduler periodically assigns new energy budgets to the guests, energy budgets of physical CPUs can be kept by using guest-level scheduling. It requires a well distribution algorithm or the assignment of only one virtual CPU of exactly one virtual machine per physical CPU. The guest-level scheduler then balances the energy consumption of physical CPUs (without knowing this fact), as it balances the one of the virtual CPUs.

This however, can also be considered as the basic limitation of guest-level scheduling. It only can fulfill energy-aware scheduling for physical CPUs if the virtual machine scheduler support the correct distribution of the correct energy budgets and supports the required assignment of one virtual CPU to one physi-

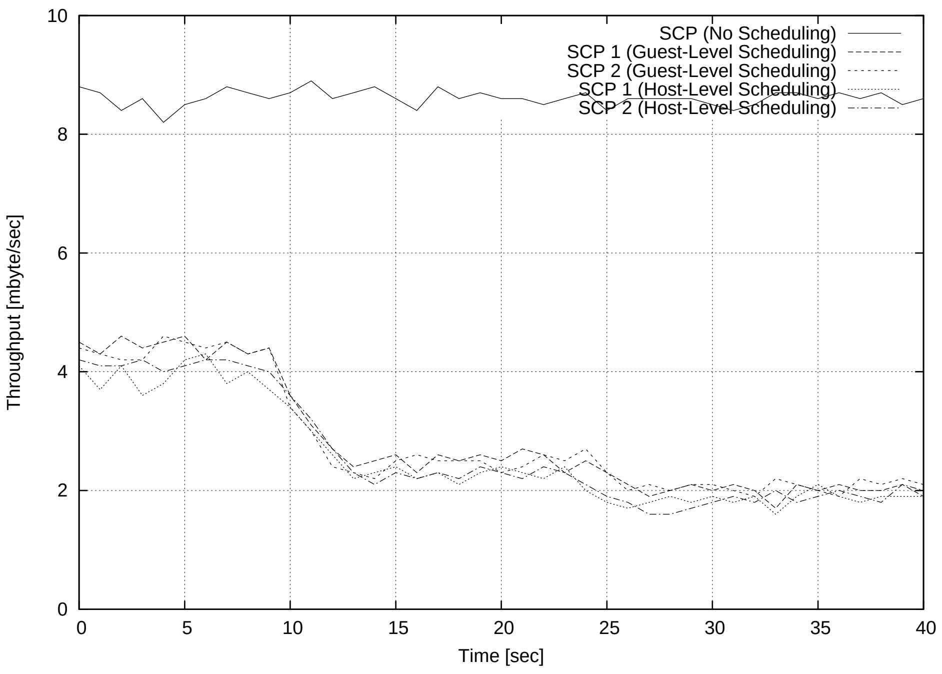

Figure 5.16: Network throughput: Now two scp instances are competing for computing power. Then guest-level scheduling and host-level scheduling provide nearly the identical results for the network throughput (the ones for host-level scheduling are still a little bit less). The throughput has been measured for an average power consumption of 25 watt (no limitation), 15 watt and 10 watt.

cal CPU for all virtual machines. Therefore we state host-level scheduling is also required for successful energy-aware-scheduling of virtual machines.

Chapter 6

Conclusion

Our major goal has been the presentation of a framework which provides the required functionality to schedule energy-aware in virtual machine environments and to support multiprocessor systems. Our solution introduced an approach which separates the energy-aware scheduling process into two levels: Guest-level scheduling and host-level scheduling. Thereby at first, we get the possibility to benefit from scheduling relevant informations available only on one of both levels. Furthermore we can enforce energy-aware scheduling using the host-level scheduler in case a guest does not support energy-aware scheduling. Finally an energy-aware scheduling policy can rely on fine-granular scheduling decision available in the guest.

Initially we stated that there is no real approach solving this topic. Some contributions are available; none of it addresses the complete issue. Therefore this thesis offers a framework based upon a collection of mechanism which achieve our goal. We introduced guest-level scheduling in order to schedule fine-granular and to keep energy-budgets with a high accuracy. Energy profiles and energy budgets offer to control the energy consumption by the guest itself. Suspension and migration allows a guest to react on its energy consumption properly. Furthermore we presented host-level scheduling which is capable to schedule complete virtual machines due to energy budgets defined per virtual machine and per physical CPU. We came up with virtual machine suspension to offer the safety in a way that a virtual machine can be enforced to keep its energy budget. We demonstrated that virtual machine migration is required to exhaust the capacities of a multiprocessor system. Our complete design is set up in way it provides the flexibility to choose where to implement the scheduling policy: In the guest-level scheduler or in the host-level scheduler. Finally our design has been proven to be very efficient concerning its additional energy consumption.

We consider our work as the foundation for future work in the sector of energy-aware processor management for virtual machine architectures. Energy-aware scheduling is the approach to overcome the issues classical mechanisms like frequency scaling can not offer for virtual machines. We believe that our framework can be used as a starting point to transfer existing energy-related scheduling policies, like temperature-aware scheduling, to virtual machine environments. Other classic scheduling policies can be utilized to distribute energy budgets in different ways, for example lottery scheduling using its tickets. As we did not offer a complete solution for multimedia systems and real time systems further work can try to elaborate mechanisms required to overcome overload situations and to define scheduling deadlines besides energy budgets. As our focus was set to a para-virtualized architecture, when we implemented our prototype, future work could try to evaluate how to port our approach to pure-virtualized or pre-virtualized architectures. Currently we do not see any problems to this issue, nevertheless. Further issues to explore are the collaboration of guest-level scheduling and host-level scheduling and the partitioning of tasks in order to get a working energy-aware scheduler. Thus mechanisms for the synchronization of the scheduling decision on both levels on the one hand and a more simplified process to decide, where to implement a part of energy-aware scheduling, on the other hand are still open positions. Finally we propose to consider how to achieve a more efficient base mechanism for virtual machine migration in order to get a better migration duration. As result the host-level scheduler could rely on lower scheduling periods and thus achieve an even more precise balancing of the energy consumption of physical CPUs.

List of Figures

Bibliography

[1] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, Berkeley, CA, February 22–25 1999. Usenix Association.

[2] 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.

[3] Liuz Andre Barroso. The price of performance. *ACM Queue*, 3(7), September 2005.

[4] Frank Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Germany, June 29 2001.

[5] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[6] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines, 2005.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM Press.

[8] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[9] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines.

In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[10] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The l4ka vision, April 2001.

[11] Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[12] Andreas Merkel. Balancing power consumption in multiprocessor systems, September 30 2005.

[13] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the ACM SIGOPS EuroSys Conference, Leuven, Belgium, April 18-21 2006*.

[14] Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-driven thermal management in smp systems. In *Second Workshop on Temperatur-Aware Computer Systems (TACS'05)*, Madison, USA, June 2005.

[15] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *SOSP '73: Proceedings of the fourth ACM symposium on Operating system principles*, page 121, New York, NY, USA, 1973. ACM Press.

[16] Marcus Reinhardt. Cooperative, energy-aware scheduling of virtual machines, August 8 2005.

[17] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[18] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[19] Jan Stöß. Using operating system instrumentation and event logging to support user-level multiprocessor schedulers, March 24 2005.

[20] The L4Ka Team. L4 - the l4 microkernel family and friends. `http://l4ka.org/`.

[21] The L4Ka Team. L4ka virtualization project web page. `http://l4ka.org/projects/virtualization/`.

[22] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Germany, May 2005.

[23] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 6–7 2004.

[24] VMware. Vmware workstation 5.5. `http://www.vmware.com/de/products/desktop/ws_features.html`.

[25] Martin Waitz. Accounting and control of power consumption in energy-aware operating systems. Diplomarbeit, Friedrich Alexander Universität Erlangen-Nürnberg, January 2003. DA-I4-2002-14.

[26] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation (November 1994)*, pages 1–11, 1994.

[27] Xen. Xen virtual machine monitor. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`.