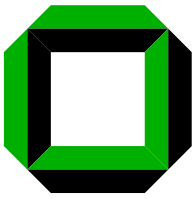


Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Diplomarbeit

Energy Accounting for Virtual Machines

Christian Lang

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa

Betreuender Mitarbeiter: Dipl.-Inf. Jan Stöß

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 Christian Lang

Abstract

This thesis presents a two-level approach to energy accounting in virtual machine environments. Instead of accounting the energy consumption of the hardware directly to the applications, as done by previous approaches, we account it to virtual machines. Each guest operating system can then obtain the energy consumption of the virtual machine and split it between its applications. Thus, energy management can leverage information intrinsic to the respective level. Furthermore, by dividing energy accounting between host-level and guest-level, we can reuse existing energy management solutions within the virtual machines. For this purpose, we introduce an energy-aware virtual machine interface, which enables the guest operating systems to estimate the energy consumption of virtual devices.

To evaluate our approach, we implemented a prototype for an existing virtual machine environment. Experiments show that the performance overhead caused by energy accounting in host-level and guest-level is less than 3.2 percent.

Contents

Chapter 1

Introduction

With the rising power dissipation of modern hardware, energy management is becoming an important issue in system design. Especially in server systems, the cost for power supply and cooling plays a major role. Limiting the energy consumption of the hardware avoids overprovisioning of power supply and cooling facilities. Furthermore, reducing the operating temperatures increases stability and reliability of the components. Most of the current hardware features built-in energy management that disables certain device features or reduces the speed during phases of low utilization. However, because the hardware is unaware of the executed software, it cannot respond to application- or user-specific requirements. Therefore, research has proposed several approaches to dynamic energy management that control energy consumption in the operating system [3, 35]. In contrast to the hardware, the operating system has enough context information to implement fine-grained energy management at the level of individual tasks.

In the last years, virtualization technology has been gaining popularity as a way of consolidating servers. Hypervisor-based virtual machine environments offer a scalable solution to host several isolated virtual machines on one physical computer. Executing multiple virtualized operating systems on one server improves hardware utilization and thus reduces cost. However, virtualized systems pose new challenges for a successful energy management. Because the virtualized operating systems are confined to virtual machines, the scope of guest-internal energy management is limited. The guest operating systems are unaware of other virtual machines and of global, machine-wide energy requirements. Conversely, energy management in the virtual machine monitor lacks the fine-grained information of the guest operating system. Similar to hardware-based energy management, the virtual machine monitor is oblivious to the applications that are encapsulated within the virtual machines.

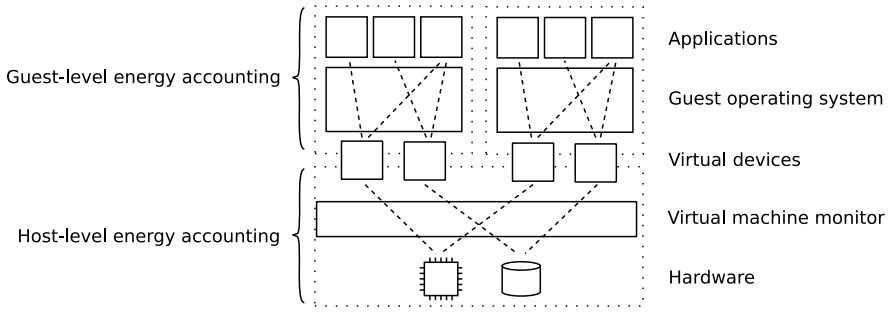


Figure 1.1: Overview of our approach. The virtual machine monitor estimates the energy consumption of the hardware and accounts it to the virtual machines. Each guest operating system obtains the energy consumption of the virtual devices and charges it to the running applications.

We argue that virtualized server systems need energy management on both layers, in the virtual machine monitor and within virtual machines. On the one hand, only the guest operating systems can implement fine-grained energy management, having regard to application-specific demands. On the other hand, only the virtual machine monitor can control global, machine-wide energy requirements and manage the energy consumption across virtual machines. This two-level approach also enables basic energy management for legacy and energy-unaware guest operating systems. Although, in this case, guest-intrinsic requirements have to be neglected, the virtual machine monitor can still control the energy consumption of the complete virtual machine. In the same way, it can enforce given energy requirements for malfunctioning or malicious guests.

1.1 Energy Accounting for Virtual Machines

Energy accounting is a prerequisite for energy management. For dynamic energy management, accounting and management form a feedback loop: energy accounting supplies the energy consumption of individual tasks to support management decisions and to control their results. For this purpose, the accounting infrastructure has to estimate the energy consumption of the hardware and account it to the tasks. In virtualized environments, however, this information is distributed between the virtual machine monitor and the guest operating systems: the virtual machine monitor knows the hardware and can estimate the amount of energy consumed by the devices, but only the guest operating systems can account the energy consumption to guest-internal applications.

This thesis presents a two-level accounting infrastructure for hyper-

visor-based virtual machine monitors. The device subsystems of the hypervisor estimate the energy consumption of the hardware and apportion it between virtual machines. The guest operating systems further split the energy consumption between the applications. In order to enable guest-level energy management, the guest operating systems must be able to estimate the energy consumption of virtual devices. However, virtualized hardware does not adhere to the assumptions made by classical approaches and is therefore not accounted correctly. Thus, the virtual machine monitor simulates energy-related behavior of real hardware to enable correct accounting of virtual devices. Alternatively, we install paravirtualized device drivers within the guest operating systems, that query the energy consumption directly from the virtual machine monitor. Figure 1.1 gives an overview of our approach.

To evaluate our solution, we implemented a prototype for the L4Ka virtual machine environment, a virtualization framework based on the L4 microkernel. We extended the virtual machine monitor and a user-level disk driver to support energy accounting. Furthermore, to demonstrate guest-level accounting, we adapted an existing energy-aware operating system to execute within our virtualized environment. Performance measurements show that the worst-case overhead caused by the accounting infrastructure is about 3.2 percent.

1.2 Structure of this Thesis

The remainder of this thesis is structured as follows. Chapter 2 provides background information and relates this work to previous research in the fields of virtualization and energy management. Chapter 3 presents the design of our accounting infrastructure. Chapter 4 describes the implementation of our prototype, which we use for the evaluation in Chapter 5. Finally, Chapter 6 summarizes our approach and points out directions for future work.

Chapter 2

Background and Related Work

Our solution brings together two fields of research: virtualization and energy accounting. In this chapter we will provide background information and discuss previous work in both fields. First, we will discuss the design of and, in particular, the differences between existing virtualization environments. Afterwards, we will present several approaches to resource accounting and and how they are used to account energy consumption.

2.1 Virtualization

Many virtualization environments are available, using different approaches. We will look at three areas relevant to our design, where virtualization environments implement different solutions, namely the structure of the virtual machine monitor[1], the applied virtualization technique, and the virtualization of I/O devices.

2.1.1 Virtual Machine Monitor Structure

We can distinguish two basic structures of virtual machine monitors: hosted and hypervisor-based. The hosted monitor runs on an existing operating system, the *host*. It can leverage the infrastructure (e.g. the device drivers) of the host operating system but is also limited to the constraints set by the host. Because the virtual machine monitor runs as a user-application on top of the host operating system, it does not have full control over the system resources, but is subject to the CPU and resource scheduling policies of the host. Available hosted solutions include VMware Workstation and VMware Server (formerly: VMware GSX Server).

[1] In this thesis, we will use the term *virtual machine monitor* to denote the complete software stack providing the virtual machine abstraction.

Hypervisor-based solutions do not require a host operating system. Instead, a small *hypervisor* runs directly on the hardware. The hypervisor has full control over the hardware and decides freely how to subdivide it between the virtual machines. This leads to a better performance and scalability of hypervisor-based systems [13], which renders them more suitable for server systems. However, while hosted virtual machine monitors can take advantage of the energy management features of the host operating system, existing hypervisor-based solutions, such as L4Linux [28], Xen [7], and VMware ESX Server do not implement any energy management.

2.1.2 Platform Virtualization

Apart from their structure, virtual machine monitors differ in the used virtualization technique. We can distinguish two different approaches, full virtualization and paravirtualization, that virtualize the underlying architecture to a different extent.

Full virtualization faithfully simulates the architecture expected by the guest operating system. It allows for running unmodified operating systems within virtual machines. To ensure that the guest operating systems are isolated from each other, the virtual machine monitor has to discover instructions that could affect other virtual machines, so-called *sensitive instructions*, and emulate them appropriately. This procedure is costly and degrades the performance of virtualized guest operating systems. Existing solutions that implement full virtualization include Microsoft Virtual PC and the virtualization environments by VMWare.

Paravirtualization eliminates the performance problems of full virtualization by executing the guest operating systems cooperatively. Instead of isolating virtual machines at hardware-level, paravirtualization provides isolation by modifying the source code of the guest operating systems appropriately. The guests use an extended hardware interface, in place of the original architecture, that includes calls to the virtual machine monitor. These *hypercalls* are used to circumvent sensitive instructions. In contrast to full virtualization, paravirtualization offers performance close to native hardware. The price for paravirtualization, however, is high: each guest operating system has to be ported to the extended hardware architecture manually. Furthermore, paravirtualization is only possible for operating systems whose source code is available. Nevertheless, paravirtualization has recently gained much public interest, especially in conjunction with Linux as guest operating system. Virtual machine environments implementing paravirtualization are, for example, L4Linux and Xen.

2.1.3 I/O Device Virtualization

The virtual machine monitor not only has to virtualize the CPU, but also all the devices that shall be shared between virtual machines. For this purpose, it provides one device driver that has exclusive access to the real device. Every device access in the guests is redirected to this driver. Hosted models can use the device drivers of the host operating system. In contrast, hypervisor based models have to provide their own device drivers (e.g. ESX Server) or reuse the drivers of guest operating systems (e.g. Xen [10], L4Linux [16]). To reuse a device driver within a virtual machine, the hypervisor grants pass-through access for the device to the respective guest operating system. Within this operating system, an additional software component exports the device's functionality to other virtual machines. This approach complicates energy accounting, because the accounting infrastructure has to consider energy consumption of devices which are managed by distributed subsystems.

Similar to CPU virtualization, the virtual machine monitor either fully simulates real devices, reusing the existing device drivers in the guest, or it installs paravirtualized device drivers that cooperate with the virtualization environment. Again, the paravirtualized approach offers better performance but requires much effort to support new guest operating systems. Simulating real devices allows for running guest operating systems out of the box, without installing custom device drivers. Furthermore, this approach supports legacy operating systems whose internals are unknown.

Using custom device drivers in the guest is not limited to environments that apply paravirtualization. It is common practice to install paravirtualized device drivers in fully virtualized guests (such as Windows running on VMware). Although no modifications to the guest's source code are required, custom device drivers can be considered as paravirtualization, because the guest kernel is effectively modified.

2.2 Energy Accounting

Energy management requires on-line energy accounting that supplies information about the energy consumption of individual tasks. We will discuss energy accounting in two parts. First, we will look at different approaches to resource accounting in general, then we will discuss how existing solutions apply this to account energy consumption.

2.2.1 Approaches to Resource Accounting

For effective resource management it is critical to charge resource usage to the *resource principal* that caused it. By resource principals, we denote all entities in the system that consume resources. For example, if two applications interact in a client/server relationship, the client should be charged for the resources the server uses while handling the requests. Classical operating systems usually account resource usage to threads or processes (protection domains). Thus, services located in other protection domains are not accounted correctly. Requests are scheduled within the server's scheduling context, with the server's priority, and resource consumption is charged to the server.

Vertical Structuring

One way to avoid the aforementioned problem is to structure the operating system vertically: a vertical structured system abandons shared services and executes as much as possible within the protection domain of each application. This approach has first been proposed by the designers of Nemesis [14]. Nemesis is a research operating system designed from scratch to support multimedia applications. It provides resource accountability of applications by multiplexing all resources at a low level. Nemesis is based on a microkernel, but unlike other microkernel-based systems it does not build on shared servers. Protocol stacks and most parts of device drivers are implemented in user-level libraries.

Hypervisor-based virtual machine environments are structured similarly. The hypervisor also multiplexes the system resources between the virtual machines at a low level. Each virtual machine uses its own protocol stack and services. Unfortunately, one exception is the usage of I/O devices. Because only one device driver can use the device exclusively, all guests share a common driver provided by the virtual machine monitor. To process requests, this shared device driver causes CPU cost, which is not accounted to the virtual machines. Recent experiments [5] showed that the CPU overhead for I/O processing is substantial and cannot be neglected. However, the problem of I/O emulation overhead is not limited to CPU cost. In the same way, the shared device drivers can use other devices to fulfill requests, causing additional cost that has to be accounted to the originating virtual machine. As an example, the virtual machine monitor can simulate a local disk drive to the virtual machines, but store data on a network attached storage. In this case, the cost for the network usage have to be charged to the correct virtual machine as well.

Thus, the vertical structure of virtual machine environments is not sufficient to ensure accurate resource accounting. Because the virtual

machines still share common device drivers, we need additional mechanisms that allow us to account for interaction between different protection domains.

Thread Migration

Thread migration as described in [9] is a first step to decouple resource accounting from protection domains. Threads are split into the *execution context* and the actual *thread of control*. An execution context represents the context in which a thread can run and is bound to a protection domain. The thread of control, however, can migrate between execution contexts that are located in different protection domains. Resource usage is accounted to the thread of control. During client/server interaction the thread of control *migrates* into the servers protection domain and executes server code. Thus, all costs are automatically accounted to the client that initiated the request.

Although thread migration brings better accountability for CPU-bound services, it is not suitable for accounting of I/O devices. Most devices are interrupt-driven and thus work asynchronously to the current thread of control.

Resource Containers

Thread migration still keeps a static mapping between threads and resource principals. Banga and colleagues [1] propose to completely decouple the resource context from existing operating system objects. They represent resource contexts by a new abstraction: *resource containers*. A resource container contains all resources being used by a certain activity. It can be bound dynamically to processes and other kernel abstractions. This *resource binding* notifies the kernel about who is to be charged for the cost caused by the process.

The concept of resource containers overcomes the limitations of thread migration by allowing services to switch between different resource containers independent of thread switches; if a device driver receives an interrupt, it switches to the resource container that is responsible for the device activity. However, resource containers are designed for monolithic kernels that have full control over all resources. In a virtual machine monitor, where the device subsystems can be distributed between different protection domains or even virtual machines, resource containers are not applicable.

2.2.2 Accounting of Energy Consumption

To implement fine-grained energy accounting, the operating system has to break down the overall energy consumption of the hardware to the level of single tasks. This is complicated by the extensive parallelism in the hardware and the operating system. The CPU and other devices are time-multiplexed between tasks, and most devices handle multiple request simultaneously, asynchronously to the CPU. For example, the disk may handle a request by a task *A*, while the network card is processing a packet sent by task *B* and the CPU is executing a third task *C*.

In order to attribute energy values to individual tasks, the operating system has to measure the energy consumption for each device individually and with a high temporal resolution. Today's hardware offers no direct way to query energy consumption and installing adequate measurement instruments is too expensive. Thus, existing approaches monitor the visible behavior of devices to estimate their energy consumption. In the remainder of this section, we will look at different solutions that implement fine-grained energy accounting.

ECOSystem

ECOSystem [35] is a modified Linux kernel that manages the energy consumption of mobile systems to extend their battery lifetime. It uses resource containers for accounting. The ECOSystem kernel estimates the energy consumption of tasks based on an energy model of the system. It attributes power consumptions to the different states of each device (e.g. *standby*, *idle*, and *active*) and measures the time a device spends in the respective state. If a task causes a device to switch to a higher power state, the kernel charges the task with the additional cost.

The presented *currentcy* model allows to manage the energy consumption of all devices in a uniform way. Having estimated the energy consumption, the kernel does not distinguish between the cost contributed by different devices; all cost that a task causes is accumulated to one value. This allows the kernel to control the overall energy consumption of the system without considering the currently installed devices. However, this renders the approach unsuitable for energy management schemes such as thermal management which have to control the energy consumption of individual devices.

Event-Driven Energy Accounting

Bellosa and colleagues [3] propose to estimate the energy consumption of the CPU for the purpose of thermal management. Also based on resource containers, the approach leverages the performance monitoring

counters present in modern processors to accurately estimate the energy consumption caused by individual tasks. However, here the estimated energy consumption is just a means to an end. Based on the energy consumption and a thermal model, the kernel estimates the temperature of the CPU. If the temperature reaches a predefined limit, the system can throttle the execution of individual tasks according to their energy characteristics.

Merkel et al. [19] apply the same mechanisms to balance the power consumption (and thus the temperature) of processors in SMP systems. They propose to analyze the energy characteristics of processes to identify *hot* tasks; that is, tasks that cause a high energy consumption on the processor. The scheduler then allocates the hot tasks equally among the CPUs to avoid thermal imbalances between them. This prevents the individual CPUs from overheating and thus reduces the need for throttling.

Chapter 3

Design

Previous work has proposed several approaches for energy accounting, including solutions for server systems [3, 19]. All approaches are designed for monolithic kernels and assume full knowledge of the system resources and the resource principals. In virtualized systems, however, this information is distributed: On the one hand, only the virtual machine monitor has access to the hardware and can estimate the energy consumption of the devices. On the other hand, only the guest operating systems have fine-grained information about the applications that are running on the server.

We propose an energy accounting infrastructure for hypervisor-based virtual machine monitors that complements existing guest-internal approaches. In order to enable energy accounting in the guest operating systems, the virtual machine monitor estimates the energy consumption of the physical devices and accounts it to virtual devices. The guest operating systems obtain the energy consumption of the virtual devices and account it to their resource principals.

In addition, the accounting infrastructure provides accounting information to the *resource monitor*, a component within the virtual machine monitor that implements host-level energy management. The resource monitor controls global, machine-wide energy requirements and enforces given requirements for energy-unaware or uncooperative guest operating systems.

3.1 Basic Architecture

The ultimate goal of energy management is to control the energy consumption of the hardware. Thus, the first step for energy accounting is to estimate the energy consumption of the physical devices. Most of the current hypervisor-based virtual machine monitors reuse device drivers

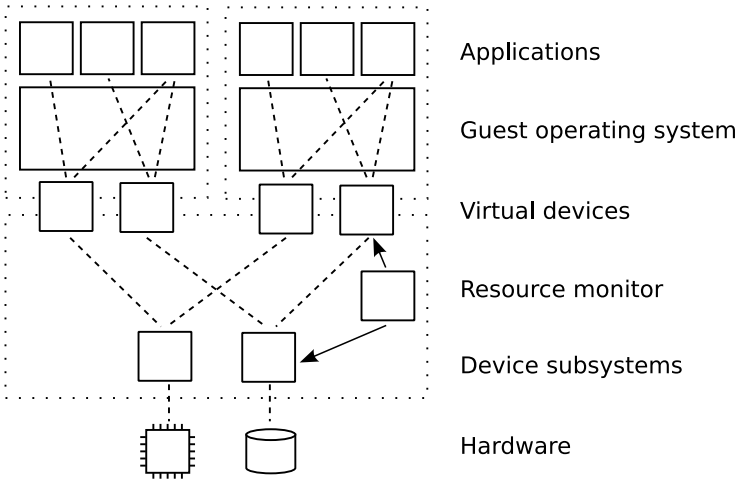


Figure 3.1: Overview of the accounting architecture. Accounting components in each device subsystem estimate the energy consumption of the device and apportion it between the virtual devices. The virtual devices export their energy consumption to the guest operating system, which charges it to its resource principals. Based on the accounting information provided by the device subsystems, the resource monitor controls the energy consumption of physical and virtual devices.

located within virtual machines [10, 16]. Hence, the knowledge required for energy estimation is distributed in the system. Therfore, we estimate the energy consumption of each device in a separate accounting component, located with the corresponding device subsystem. Due to the vast number of different devices and device drivers, it is important to be able to adapt to new devices easily. Thus, our solution will implement energy estimation with minimal changes to the driver that is managing the device.

In order to apportion the energy consumption of physical devices correctly between the virtual machines, the virtual machine monitor has to assign each device activity to the originating virtual machine. In particular, we have to account for interaction between different devices. For example, the disk driver uses the CPU to process requests. Moreover, it may even need the network to transparently access data located on network attached storage. Unlike existing accounting facilities, such as resource containers [1], we cannot assume a monolithic kernel that comprises all device subsystems. Because of the distributed nature of our accounting architecture, we have to account for device interaction across protection domains. For this purpose, we augment the normal request data with accounting information and charge energy consumption recursively between the various device subsystems.

The guest operating system is unaware of the real devices and their

energy consumption. Its energy accounting is based solely on the accounting information provided by the virtual machine monitor. To be able to account energy consumption to individual resource principals, the guest operating system has to know the cost separately for each virtual device (see Section 2.2.2). The virtual machine monitor has to map the estimated energy consumption of physical devices to the virtual devices in a way that enables effective guest-level energy management. We will define an energy model for virtual devices that satisfies the demands of energy management in guest-level and host-level and enables a wide range of policies.

Finally, to enable guest accounting, we have to import the energy consumption of virtual devices into the guest operating system. We can do so by installing paravirtualized device drivers into the guest operating system. Knowing internals of both levels, the driver queries the energy consumption from the virtual machine monitor and accounts it to the guest-internal resource principals. However, we also want to support energy accounting in fully virtualized guests. For this purpose, we simulate the energy characteristics of real devices to enable energy estimation within the virtual machine.

Figure 3.1 gives an overview of our approach. In the remainder of this chapter, we will discuss the different parts and aspects of our solution in more detail. We will first discuss energy estimation for physical devices (Section 3.2). We will then define an energy model for virtual devices (Section 3.3). Afterwards, we will discuss how we account for device interaction in our distributed accounting infrastructure (Section 3.4). Finally, we will present methods to import the energy consumption of virtual devices into the guest operating system (Section 3.5).

3.2 Per-Device Energy Estimation

Energy estimation and accounting requires detailed knowledge about the device. This includes static information in form of an energy model of the device, as well as runtime information such as the current power state and the clients that are currently using the device. The runtime information of a device is only available to the subsystem that is managing the device (unless we have a monolithic kernel). Thus, we implement energy accounting directly within each device subsystem. It estimates the energy consumption of the device and apportions it between the clients that use the device.

Despite these close links between the device driver and energy accounting, we want to reuse existing drivers with minimal modifications. Because of the multitude of different devices and device drivers, it is

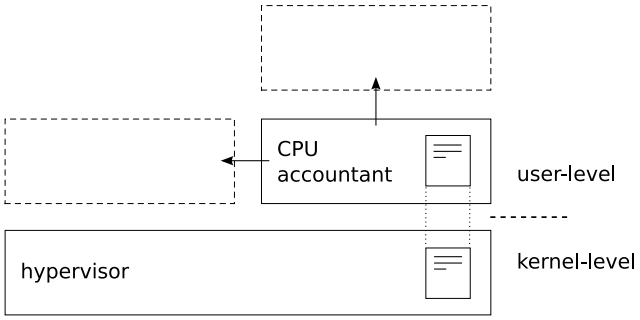


Figure 3.2: User-level accounting of CPU energy consumption. The priviledged hypervisor writes information about the CPU into a log file shared with a user-level CPU accountant. The CPU accountant analyzes the log entries and charges the energy consumption of the CPU to other components (e.g. to user-level device drivers or virtual devices).

necessary to be able to add accounting support to new drivers easily. Below, we propose two ways to achieve this.

3.2.1 Unmodified Driver Reuse

In [16], LeVasseur and colleagues present a flexible method to reuse unmodified legacy device drivers by executing them inside virtual machines. They export the device drivers functionality by means of a separate translation module that mediates requests between the device driver and external clients. The translation module runs in the same address space as the device driver and handles all requests sent to and from the driver. Consequently, it has access to all information relevant for accounting. We can implement accounting completely in the translation module, without changing the original device driver.

3.2.2 Instrumentation and Event Logging

In some cases, implementing energy accounting in the same component that is controlling the device is undesirable or unfeasible. For example, implementing energy accounting in the privileged hypervisor affects system reliability and flexibility. However, the CPU and possibly other devices are managed directly by the hypervisor. To decouple energy accounting from such critical components, we use system instrumentation and event logging as presented in [25]. To instrument existing components with log handlers, we still have to modify their code. However, the modifications are uncritical and do not change the original behavior of the component.

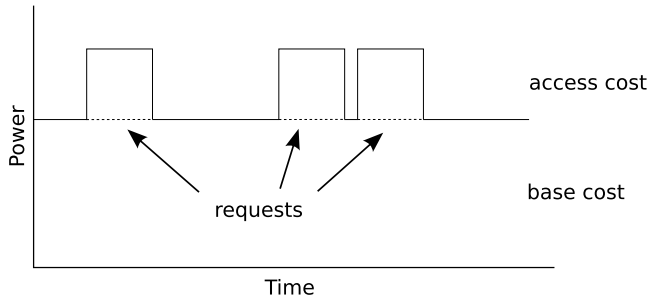


Figure 3.3: Simplified power consumption of a device, split into base cost and access cost.

In our case, we instrument the critical component to record relevant information for the respective device into an in-memory log file. The log file is shared with a user-level accounting component, which periodically analyzes the data. This user-level accountant estimates the energy consumption of the device and accounts it to the clients, based on the data contained in the log file. In Figure 3.2, we depicted how event logging can be used to account CPU energy consumption.

3.3 Virtual Energy Model

Virtualization introduces an additional layer of indirection between the operating system and the hardware. In contrast to real hardware, virtual devices have side effects on more than one physical device. This leads to a more complex energy model for the virtual devices. The energy consumption of one virtual device comprises energy consumption of several physical devices. In this section, we will define an energy model for physical and for virtual devices and discuss how we map the energy consumption of the hardware to virtual devices.

As a first step, we break down the energy consumption of each physical device into base cost and access cost. *Base cost* is the minimum power consumption of the device; the amount of energy the device needs even if it is idle. This portion of the cost cannot be influenced by the operating system. *Access cost* is the additional energy consumption that is caused by clients using the device. For energy management, access cost is the more important part. It is the portion of the energy consumption that the operating system can influence by controlling device activity. Because of this fundamental difference between base cost and access cost, energy management, to be effective, has to be able to separate the two. Figure 3.3 illustrates the concept of base cost and access cost.

3.3.1 Access Cost

Access cost is the portion of the devices' energy consumption that can be attributed to individual activities which are executed within the virtual machines. To enable the guest operating systems to charge energy consumption to the correct resource principal, we must account all energy consumption to the virtual device that initiated a request. We can do so transparently, by adding the cost of all involved devices to one value. This accumulated value is, however, not sufficient for all applications. Thermal management, for example, needs to control each *physical* device separately. If we transparently add all cost involved in a disk operation and charge it as disk energy consumption to the virtual machine, the guest operating system does know neither the real energy consumption of the disk nor the real energy consumption of the CPU. Thus, we have to to pass on the energy consumption of all physical devices explicitly.

3.3.2 Base Cost

Although the base cost plays a minor role for energy management, its correct handling is required for some applications: the resource monitor has to know the base cost to be able to calculate the total energy consumption of the system. In addition, the total energy consumption is needed for monitoring or billing of device usage on host-level and guest-level.

In contrast to the access cost, the base cost of a virtual device does not include base cost of different physical devices. Each device driver calculates the base cost of its device and apportions it to the virtual machines, for example by dividing it equally between them. Each guest operating system further splits its portion of the cost between the guest-internal resource principals.

Multiple Power States

Most devices have several sleep modes that disable certain device activity and thus reduce idle cost. Suspending devices, however, is unrealistic for server systems because current devices need too long to wake up from sleep modes and thus would cause unacceptable latencies. Especially disk drives need several seconds to wake up, because they have to spin up to full speed before they can handle requests. Furthermore, on a reasonably utilized server, the phases where a device is idle are too short to amortize the additional cost that is caused by the transitions between different power states.

A promising approach for server systems is having multiple *active* modes instead of *sleep* modes. In an active power saving mode, the

device can handle requests with lower speed and does not have to wake up. Recent server systems already use processors that support *frequency scaling* to reduce power dissipation. A similar approach for disks has been presented in [11]. In this paper, Gurumurthi and colleagues propose multi-speed disks that allow for running disks at lower spinning speed during phases of low disk utilization.

To account for multi-speed devices, we pass on changing base cost to the virtual devices. We propose to decouple the current power state of the physical device and the virtual devices to enable fair accounting of base cost. Virtual machines that do not use the device are charged for the lowest active power state. Higher base costs are only charged to the virtual machines that are actively using the device.

3.4 Distributed Accounting

With resource containers, all accounting information is stored in global data structures. To charge cost to a resource container, device drivers just have to add it to the respective counter variable; the changes are immediately visible to every other device. In our model, the accounting components reside in different protection domains and do not share global data structures. Thus, accounting information is distributed between the devices. To share this information, the device subsystems have to export it explicitly through interfaces or shared memory.

3.4.1 Recursive Request-Based Accounting

To correctly account for interaction between different devices, we charge energy consumption recursively. If one device subsystem uses other physical devices to fulfill a request, it charges these additional costs to its client. This way, we propagate the energy consumption of all physical devices involved in an I/O operation to the corresponding virtual device.

If the device subsystems are distributed in the virtual machine monitor, they already have interfaces through which they communicate. Instead of introducing a new interface, we add cost information to the normal request data. As described in Section 3.3.1, the cost information of different devices must not be accumulated to one value, but has to be transferred separately. Thus, each request explicitly carries the energy consumption it has caused on each device.

As an example, we regard a shared disk driver which uses the network to transfer data to network attached storage. The network driver estimates the energy it consumed when processing the network packages and returns it to the disk driver. The disk drivers adds the cost of

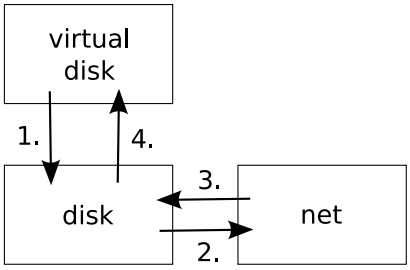


Figure 3.4: Recursive accounting of energy consumption. The virtual disk sends a disk request to the shared disk driver (1.). To fulfill the request, the disk driver uses the network driver (2.), which estimates the cost for processing the request and returns it to the disk driver (3.). Finally, the disk driver charges the energy consumption of the disk and the network interface to the virtual disk (4.).

all network packages used to transfer a disk block and charges it to the virtual disk, together with the disk energy consumption and the CPU energy used to process the disk request. The procedure is depicted in Figure 3.4.

3.4.2 Energy Meters

The base cost of a device cannot be attributed clearly to requests. To account these cost, each device provides an virtual electricity meter, or *energy meter*, for each virtual machine. It indicates the virtual machine's share in the total energy consumption of the device (including the cost already charged with the requests).

Another case where request-based accounting is impossible is the CPU. The CPU cost cannot be attributed to requests, because the CPU does not work request-based. Rather, the kernel usually gives each thread the illusion of running exclusively on a CPU and switches transparently between different threads. Thus, we also use energy meters to account CPU cost. CPU accounting provides a meter for each virtual machine and each device. Other accounting components which use the CPU can query the meter each time they switch between clients to determine their respective energy consumption.

3.4.3 Global View

To implement global energy management policies, the resource monitor needs a global view of the accounting data; it has to gather the data that is distributed in different devices. In contrast to the request-based accounting for virtual devices, the resource monitor only needs coarse-grained information to support system-wide decisions.

We propose to export the required accounting data via shared memory. That way, the resource monitor can access detailed information about the energy consumption of virtual devices without additional communication cost. The resource monitor shares a separate memory region with the accounting component of each physical device. Through the shared memory region, each device exports accounting information for each virtual device. To provide current information to the resource monitor, the devices have to keep the values stored in shared memory up-to-date. Immediately after a device completed a request and estimated the consumed energy, it updates the respective value in the shared region.

3.5 Guest Accounting Support

So far, we discussed how the virtual machine monitor accounts energy consumption to virtual devices. To enable guest-level energy management, we have to import the energy consumption of virtual devices into the guest operating system.

As described in Section 2.1, virtual machine monitors use two different approaches to virtualize devices: emulating standard devices or installing custom device drivers in the guest. Emulating a standard device allows for keeping the original, unmodified device driver of the guest. The virtual device and the guest communicate only via the hardware interface. Thus, the guest operating system can use the same driver in a virtual machine and on real hardware. In contrast, custom device drivers use a paravirtualized device interface and communicate directly with the physical device driver. We will discuss each of the two approaches separately and suggest energy accounting support that preserves their respective goals and advantages.

3.5.1 Energy-Aware Device Emulation

Current hardware offers no direct way to query energy consumption. Instead energy estimation uses certain device characteristics, which correlate to the energy consumption of the device. By emulating the according behavior for the virtual device, we support energy estimation in the guest without modifications to the guest's energy accounting (that is, if the guest operating system implements energy accounting, which is not the case for most current operating systems). For example, we can virtualize the processors performance counters to support event-driven energy accounting [3] in the guest operating system[1], or simulate different power states to support the energy model applied by ECOSystem.

[1] We follow this approach in our prototype implementation. See Section 4.3 for more information

Energy estimation is based on parameters that depend on the exact device model. For example, the energy consumption of a disk can be estimated based on its power consumption in idle and active mode and the time it remains in active mode to handle a request. Before the estimation yields accurate results, it has to be calibrated for the respective device. Thus, it supports only a predefined set of devices that have been measured beforehand. To support energy estimation for virtual devices, we have to notify the guest operating system about their respective energy parameters. Therefore, we propose to make these energy parameters configurable by a user-level application. The application can look up a table with supported devices to set up energy accounting or – in our case – query the virtual machine monitor for the correct parameters.

Because we have to translate the energy consumption of devices from our energy model to the parameter-based model of the respective guest operating system, the accuracy of the energy estimation in the guest is limited. The guest does not get the energy consumption for every request. Instead, the virtual machine monitor has to pre-estimate the energy consumption of the devices, including maintenance cost, to calculate the energy parameters of the virtual devices. Because maintenance and idle cost for a device are shared between the virtual machines, the parameters can change if the number of virtual machines on the system changes. To ensure accuracy in the long run, the guest has to ask the virtual devices regularly for updated parameters.

Accounting I/O Emulation Overhead

Another limitation, if we simulate the energy behavior of a real device, is that we cannot account for emulation overhead properly. The guest operating systems are unaware of side effects between the devices. We will discuss two methods to notify the guest operating systems about emulation overhead, without changing the guest's device drivers.

Ony way is to charge emulation overhead transparently, together with the cost of the respective virtual device. For this purpose, the virtual device increases the energy parameters of the device to cover cost of other physical devices as well. The guest operating system does not have to be aware of the CPU usage of the disk; it is automatically accounted to the right resource principal. However, this solution has one drawback: the guest does not know the energy it really has consumed on the physical CPU, because part of the CPU energy consumption is charged as disk I/O cost. This renders this approach unsuitable for certain applications, such as thermal management (see Section 3.3.1).

Alternatively, we can use *energy ballooning* to inform the guest operating system about emulation overhead. Similar to the concepts of mem-

ory and time ballooning presented in [31] and [29], energy ballooning allows us to inject artificial cost into the guest operating system, without modifying the guest's accounting code. For example, to account CPU overhead, we create a balloon thread in the guest operating system and assign the CPU overhead cost to it. Similarly, we can use dummy network packages to account for network overhead. Energy balloons allow us to inform the guest about its total cost for a specific device; however, the guest is unable to attribute emulation overhead to the originating resource principal.

None of two approaches satisfies the requirements of all energy management policies. We have to choose the more adequate approach depending on the desired goals. If the guest has to know the energy consumption of each physical device separately, we have to use energy balloons, otherwise, accounting emulation overhead transparently with each virtual device is to be preferred, because it allows the guest operating system to charge the cost to the originating resource principals and is less intrusive.

3.5.2 Paravirtualized Device Drivers

Paravirtualized device drivers are not confined to the virtual hardware interface and communicate directly with the virtual machine monitor. They know about both sides: the physical devices and the guest-internal resource principals. Therefore, in this case, accounting energy consumption to the guest's resource principals is straightforward: The device driver queries the cost directly from the physical device subsystems and charges it to the right resource principals.

Because the device driver has access to the guest's accounting system, it does not have to account emulation overhead transparently but can charge all cost explicitly to the right resource principal. However, this implies that the driver is able to account the according cost. If it is not, we have to fallback to one of the methods described before.

Chapter 4

Implementation

To evaluate our solution, we implemented a prototype for an existing virtual machine environment. In addition to the accounting infrastructure in the virtual machine monitor, we implemented an energy-aware guest that uses the accounting information provided by the host. We support two main energy consumers: CPU and disk. Furthermore, we implemented a simple resource monitor that has access to accounting information of all devices via shared memory but does not apply any energy management policy.

Before we discuss our implementation, we will describe the components our virtual machine environment consists of (Section 4.1). The subsequent sections describe the different parts of our solution. Section 4.2 discusses our implementation of the resource monitor. Sections 4.3 and 4.4 address energy accounting for CPU and disk; both sections first describe host-level and then guest-level accounting.

4.1 Environment

We implemented our prototype for the *L4Ka virtual machine environment*. The environment consists of a user-level virtual machine monitor and a paravirtualized Linux kernel, called *L4Linux*, both running on top of an *L4 microkernel*. To offer device access to the Linux instances, the environment uses designated device driver virtual machines that host Linux device drivers and export their functionality to the other virtual machines. To demonstrate guest-level accounting, we use an extended version of L4Linux that supports energy accounting with resource containers. In the remainder of this section, we will shortly describe the different parts of our environment.

4.1.1 L4Ka Virtual Machine Environment

The L4Ka virtual machine environment [28] is an infrastructure for running virtual machines on top of L4Ka::Pistachio, the latest implementation of the L4 microkernel developed at the University of Karlsruhe. L4 offers two basic abstractions: threads and address spaces. It supports inter process communication (IPC) and flexible mapping of memory regions between address spaces.

The guest operating systems are modified Linux 2.6 kernels that use L4 as hypervisor, instead of running on real hardware. The paravirtualized L4Linux uses the abstractions and operations provided by L4 to implement Linux abstractions. Accordingly, Linux threads are mapped onto L4 threads and Linux address spaces correspond to L4 address spaces. The Linux kernel executes as a separate thread in its own address space and communicates with the user-level tasks via IPC.

Furthermore, the infrastructure comprises a user-level virtual machine monitor, running as an L4 task, that manages the Linux instances and provides additional services. The environment allows for running unmodified Linux applications and native L4 applications side by side.

4.1.2 User-Level Driver Architecture

The virtual machine environment implements the driver architecture proposed in [16]. It uses unmodified Linux device drivers running in a designated virtual machine. The drivers are encapsulated in an L4Linux instance that has pass-through access to the respective devices. To export the device to client virtual machines, the driver virtual machine contains a translation module that translates between client requests and Linux primitives. On the client side, a custom device driver receives requests and forwards them to the device driver virtual machine, using the interface exported by the translation module.

4.1.3 L4Linux with Resource Containers

As an example for an energy-aware guest operating system, we use an extended version of L4Linux that supports resource containers. The original resource container extension is described in [30]. We adapted the implementation to our paravirtualized environment and added support for disk accounting (see Section 4.4.2).

4.2 The Resource Monitor

Implementing as little policy as possible within the kernel is a central design principle for the development of the L4 microkernel that ensures its flexibility and generality. Because the resource monitor is intended to be the place for global energy management policies, we implemented it in user-level. Moreover, this allows us to add new policies at runtime by replacing the resource monitor with an updated version.

Our implementation of the resource monitor does not apply any energy management policy. Nevertheless, we provided it with on-line accounting information of all devices to show the feasibility of our approach and to obtain the energy measurements presented in Chapter 5. In order to provide accounting data for global policies, all device drivers have to register with the resource monitor. Upon registration the resource monitor maps a memory region into the device subsystem's address space. Within this shared region, the device subsystem stores accounting information for each of its clients.

All device subsystems use the same data structure `shared_t` to hold the accounting information within their shared memory region. The `shared_t` data structure contains another data structure `client_info_t` for each virtual machine, which in turn contains the energy consumption the virtual machine caused by using this device subsystem. The following code segment lists the data structures.

```
typedef struct {
  energy_t base_cost;
  energy_t access_cost[MAX_DEVICES];
} client_info_t;

typedef struct {
  client_info_t clients[MAX_VMS];
} shared_t;
```

4.3 CPU Energy Accounting

We choose an approach based on performance monitoring counters to get accurate energy consumption. The microkernel multiplexes the counters between virtual machines. At each switch between virtual machines it writes performance counter values into an in-memory log file shared with a CPU accountant in user-level and with the guest operating systems. The user-level accountant periodically analyzes the log file and calculates the energy consumption of each virtual machine. We also use

event	weight [nJ]
time stamp counter	6.17
unhalted cycles	7.12
μop queue writes	4.75
retired branches	0.56
mispredicted branches	340.46
memory retired	1.73
MOB load replay	29.96
ld miss 1L retired	13.55

Table 4.1: Used event counters and their energy contribution. The selection of events and their respective weights are according to [3].

the logged values to virtualize the performance monitoring counters at guest-level.

4.3.1 Host-Level CPU Accounting

We use *event-driven energy accounting* to estimate CPU energy consumption. In [3] Bellosa and colleagues argue that using CPU load as an indicator for energy consumption yields imprecise results. They show that on modern processors high-power tasks can consume 70% more energy than low-power tasks. Event-driven energy accounting leverages event counters embedded in modern processors to get accurate results. Originally intended for performance monitoring, the counters can be used to count energy-critical events. In contrast to real energy measurements, reading event counters is fast enough to be read at each context switch. Thus, it is possible to estimate energy consumption for each individual task.

To estimate energy consumption of a processor, we assign energy values (weights) to events. We can then count the occurrences of each event by means of the performance monitoring counters to calculate the energy consumption of the CPU. Because the available counters are specific to the CPU model, the used counters and weights vary and have to be calibrated for the CPU. We implemented event-driven energy accounting for the Pentium 4 and used the values from [3] (see Table 4.1).

Instrumentation and Event Logging

The performance monitoring counters consider all code that is executed on the CPU. As we want to estimate the cost for each virtual machine, we have to read the event counters at every switch between them. The kernel has to multiplex the counter values between virtual machines, because context switches are transparent to user applications.

As mentioned before, the microkernel should offer generic mechanisms only. Thus, we want to keep the policy for energy estimation out of the microkernel. We use instrumentation and event logging [25] to transfer the relevant counter values to a user-level CPU accounting thread. At each switch between virtual machines, L4 reads the counter registers, and records them into a log file shared with a user-level CPU accountant. By setting up the performance monitoring counters from the CPU accountant, we can choose the counted processor events from user-level.

Host-level accounting has to split cost between virtual machines. L4, however, does not know about virtual machines. The virtualization environment maps Linux processes to L4 threads, which would result in too detailed counter values emitted by the kernel. We use the concept of *accounting domains* presented in [25] to group threads belonging to a virtual machine. Accounting domains map resource principals (in this case threads) to accountable entities. The logging infrastructure does not consider context switches or any other interaction within one accounting domain, but treats all resource principals contained in an accounting domain as one.

User-Level Energy Estimation

We implemented a CPU accountant as part of the user-level virtual machine monitor, which analyzes the log file and estimates the energy consumption of each virtual machine. Every 20 milliseconds, the CPU accountant checks the log file of each virtual machine for new entries. The values calculated by the CPU accountant serve solely for global accounting because the guest operating systems, for performance reasons, do not query their energy consumption from the virtual machine monitor but also read the log file written by the microkernel.

Instead of charging the complete energy consumption of each period to the according virtual machine, we subtract the base cost and split it between the virtual machines. The time stamp counter, which is included in the values recorded by L4, together with its energy weight supplies us with an accurate estimation of the processor's idle cost. It is the only counter that advances if the CPU is idle, and it constantly advances if the CPU is utilized. Thus, the energy estimation looks as follows:

```
// calculate idle energy
idle_energy = 6.17 * counter1;

// split idle energy between domains
idle_energy /= (max_domain - min_domain + 1);
```

```
for (d = min_domain; d <= max_domain; d++) {
   clients[d].base_cost += idle_energy;
}

// calculate and charge access cost
energy = 7.12 * counter2 +
         4.75 * counter3 +
         0.56 * counter4 +
       340.46 * counter5 +
         1.73 * counter6 +
        29.96 * counter7 +
        13.55 * counter8;
clients[current].access_cost[RES_CPU] += energy;
```

4.3.2 Guest-Level CPU Accounting

The guest operating system does not charge resource usage to processes, but to resource containers [1]. To support CPU accounting, each process has a *resource binding* to a resource container. When the kernel schedules a process, it binds the process's current resource container to the CPU. Each time the current resource container of the CPU changes, the kernel estimates the energy consumption and charges it to the previously active resource container.

A switch between resource containers happens when the scheduler dispatches a process with a different resource container, when the current process is bound to another resource container, or when an interrupt arrives. As these events happen frequently, the energy estimation must not cause significant overhead. Although L4 implements very fast IPC, polling energy consumption from the CPU accountant thread via IPC causes too much overhead. Each context switch or interrupt in Linux would involve two additional context switches in L4.

To avoid these communication latencies, we also apply event-driven energy accounting directly in the guest operating system to estimate the energy consumption of the virtual CPU. For this purpose, the virtual machine monitor configures the performance monitoring counters to be readable from user-level. However, the counters consider all executed code, regardless of the virtual machines. To (para-)virtualize the counters, the host-level CPU accountant maps the log file containing the values recorded by the microkernel into the guest kernel's address space.

At each switch between resource containers, the guest operating system reads the counter registers from the processor and calculates the difference to the previously measured values. Before we calculate the energy consumption from these values, we check for new entries in the

log file. If it contains new log entries, other virtual machines have been running in the meantime; we use the recorded values to factor out the portion that is contributed by other virtual machines.

4.4 Disk Energy Accounting

Disk drives are a major contributor to the overall power consumption of servers [4, 36]. Especially in systems with several disks, the power consumption of the disks is substantial and exceeds the power consumption of the CPU. Furthermore, we need disk accounting to account for CPU overhead during disk emulation. Disk accounting differs from CPU accounting in several ways:

- The disk driver is located in user-level within a virtual machine, instead of in the privileged microkernel.
- Disk accounting has to charge cost of two physical devices: the disk and the CPU. Thus, we have to apply recursive accounting.
- Disk accounting is based on requests. We have to apportion cost between requests and transfer it with the request data.

We will first discuss energy estimation and apportionment in the user-level disk driver (Section 4.4.1). We will then describe our extensions to the resource container implementation, which allows for energy accounting per disk request within the guests (Section 4.4.2).

4.4.1 Host-Level Disk Accounting

To handle disk requests, the virtual machine environment uses unmodified Linux drivers encapsulated in a designated virtual machine. A translation module within the virtual machine receives disk requests from other virtual machines, translates them to basic Linux block I/O requests, and passes them to the original device driver. When the device driver has finalized the request, the module again translates the result and returns it to the client virtual machine. According to Section 3.2.1, we did not modify the device driver and implemented energy estimation and apportionment completely in the translation module. Because it handles all disk requests from the clients to the driver and vice versa, it has all information required for accounting.

The cost for the virtualized disk consists of the energy consumed by the disk and the energy consumed by the CPU while processing the requests. The translation module estimates the energy consumption of the disk using a simple energy model and, like the client virtual machines,

estimates the CPU energy consumption based on the paravirtualized performance monitoring counters. We separate both values into base cost and access cost; the access cost are charged with the result that is returned to the client, whereas the base cost is exported through an energy meter. The client operating system can query the energy meter via IPC.

Disk Energy

To estimate the energy consumption of the disk, we use an energy model that depends on the specific disk model. Instead of attributing energy consumption to events, we attribute power consumption to different device states. As mentioned earlier, suspending the disk is unrealistic for server systems. Thus, we do not consider sleep modes. Nevertheless, we have to distinguish two different power states: active and idle. We also need the transfer rate of the disk to calculate the transfer time of a request; that is, the time the device remains in active state to handle a request. In reality, the energy consumption of disks is more complex; to simplify energy estimation, we ignore several parameters that affect the energy consumption of requests, such as the seek time or the rotational delay, and assume constant values.

After the device driver completed a request, the translation module estimates the energy consumption of the request, depending on the number of transferred bytes:

```
// estimate transfer cost for size bytes
time = size / transfer_rate;
energy = time * (active_power - idle_power);
```

Because the idle cost are independent of the requests, they do not have to be calculated for each request. However, we have to update the cost periodically, to supply the resource monitor with up-to-date accounting data. In addition, we estimate the idle cost every time one of the guest reads its energy meter.

```
// estimate idle energy since last time
time = now - last;
idle_energy = idle_power * time;

// divide between clients
idle_energy /= client_count;
list_for_each(l, &clients) {
   c = list_entry(l, struct client_entry, list);
   clients[c->domain].base_cost += idle_energy;
}
```

CPU Energy

The translation module and the device driver consume energy when they process disk I/O requests. Moreover, they are encapsulated in a full Linux kernel, which also uses the CPU. It is infeasible to track the energy consumption of individual requests through the Linux kernel. Linux combines requests to get better performance and delays part of the processing in workqueues and tasklets. Estimating the CPU energy consumption between each request would cause too much overhead. Thus, we only estimate the energy consumption at times and apportion it between the requests.

The Linux kernel constantly consumes a certain amount of energy, even if it does not handle disk requests. According to our energy model, we do not charge this base cost with the requests. To be able to distinguish the base cost from the access cost, we approximate the base cost of the Linux kernel at boot time, before the client virtual machines use the disk.

We assume constant CPU cost per request to predict the energy consumption of future requests. Every 50 request, we estimate the CPU energy consumption by means of the virtualized performance monitoring counters and adjust the expected cost for the next 50 requests. The following code illustrates how we calculate the cost per request. In this code segment, the static variable `cpu_unaccounted` keeps track of the deviation between the consumed energy and the energy consumption that we have already charged to the clients. The function `get_cpu_energy()` returns the CPU cost since the last query.

```
// estimate base cost
time = now - last;
base_energy = cpu_base_power * time;
cpu_unaccounted -= base_energy;

// calculate cost per request
req_count = 50;
cpu_energy = get_cpu_energy();
cpu_unaccounted = cpu_unaccounted + cpu_energy
                - (cpu_req_energy * req_count);
cpu_req_energy = cpu_unaccounted / req_count;
```

4.4.2 Guest-Level Disk Accounting

The Linux kernel is optimized for performance. Especially the block device subsystem is heavily optimized to compensate the latencies of hard disks compared to CPU and main memory. The kernel caches every data

transfer to or from the disk in the page cache and delays every write-out to the disk. Unfortunately, this optimizing for performance worsens resource accountability.

In the client virtual machines, we use a custom device driver to forward disk requests to the device driver virtual machine. The custom device driver receives single disk requests from the Linux kernel. The request contains no information about the user-level application that caused it. At this stage it is impossible to find out who initiated the request.

We use resource containers to track disk requests through the page cache. For this purpose, we add a pointer to a resource container to every data structure involved in a read or write operation. When an application starts a disk operation, we bind the process's current resource container to the according page in the page cache. When the kernel writes the pages to the disk, we pass the resource container on to the respective buffer heads and `bio` structures.

The custom device driver in the client accepts requests in form of `bio` objects and translates them to a request for the device driver virtual machine. When it receives the reply, together with the cost for processing the request, it charges the cost to the resource container that is bound to the `bio` structure.

Chapter 5

Evaluation

We evaluated our solution from two different perspectives. In the first series of experiments, we measured the performance of our prototype to determine the runtime overhead caused by energy accounting. In the second series, we regarded the accounting information provided by our infrastructure to show the feasibility of our approach.

5.1 Test Environment

For our measurements, we used the L4Ka::Pistachio microkernel, the latest L4 microkernel developed at the University of Karlsruhe, and a current version of the L4Ka virtual machine environment. In all experiments we used one designated virtual machine that is hosting the disk driver and one or two client virtual machines that execute our application workload. The guest operating systems are paravirtualized L4Linux 2.6 kernels booting a Debian Woody installation from a ramdisk. We used a desktop computer with an Intel Pentium 4 CPU, running at a clock speed of 1.5 GHz, and 1.5 GB of main memory.

5.2 Performance

In order to be practical for real world applications, energy accounting must not cause a visible degradation of the system performance. Thus, we conducted several performance benchmarks within the virtual machines to measure the overhead of the extensions we made to the virtualization environment. Measuring the performance within the guest operating system reveals the effective performance of the system as it is seen by the applications running on top of the guest operating system.

5.2.1 Context Switch Latency

To determine the overhead caused by CPU accounting, we measured the context switch latency in the guest operating system. To apportion cost between individual tasks, the Linux kernel has to estimate and account energy consumption at each switch between them. Thus, the most time-consuming parts of CPU accounting are executed at context switches. Because the context switch is a critical path in the kernel which is frequently executed, CPU energy accounting must be fast.

To measure the context switching overhead, we used the *lat_ctx* benchmark, which is part of the LMbench suite. LMbench is a collection of mini benchmarks that evaluate the performance of UNIX systems. Lat_ctx measures the context switching time for a varying number of processes with varying size. The processes are connected in a ring of UNIX pipes and pass a token between each other. Before passing the token, each process touches a certain amount of memory to pollute the cache and simulate applications of different size. The LMbench script starts several runs of lat_ctx with process sizes ranging from 0 to 64 kilobytes. During each run it varies the number of processes – starting with 1 and going up to 92 processes.

To attribute the accounting overhead to individual parts of our implementation, we executed the benchmark with four different configurations. Starting with the original system that does not implement energy accounting, we add more functionality in each step. In the following paragraphs, we will shortly describe the four configurations. Each configuration includes all functionality of the previous configurations.

"none" The basic configuraton comprises the unmodified virtual machine environment without energy accounting.

"host CPU" This configuration adds host-level CPU accounting, which comprises two elements: At each switch between virtual machines, the microkernel records eight performance monitoring counters to a log file. Furthermore, the CPU accountant wakes up every 20 milliseconds to analyze the new log entries and estimate the CPU energy consumption of the virtual machines.

However, these two elements are not directly tied to guest-internal context switches. Because we execute the benchmark only in one virtual machine, the measurements do not include context switches between virtual machines, and the kernel does not record performance counter values. Accordingly, the log file contains no entries and the CPU accountant is idle as well. Host-level CPU accounting should thus not influence the guest-internal context switching times.

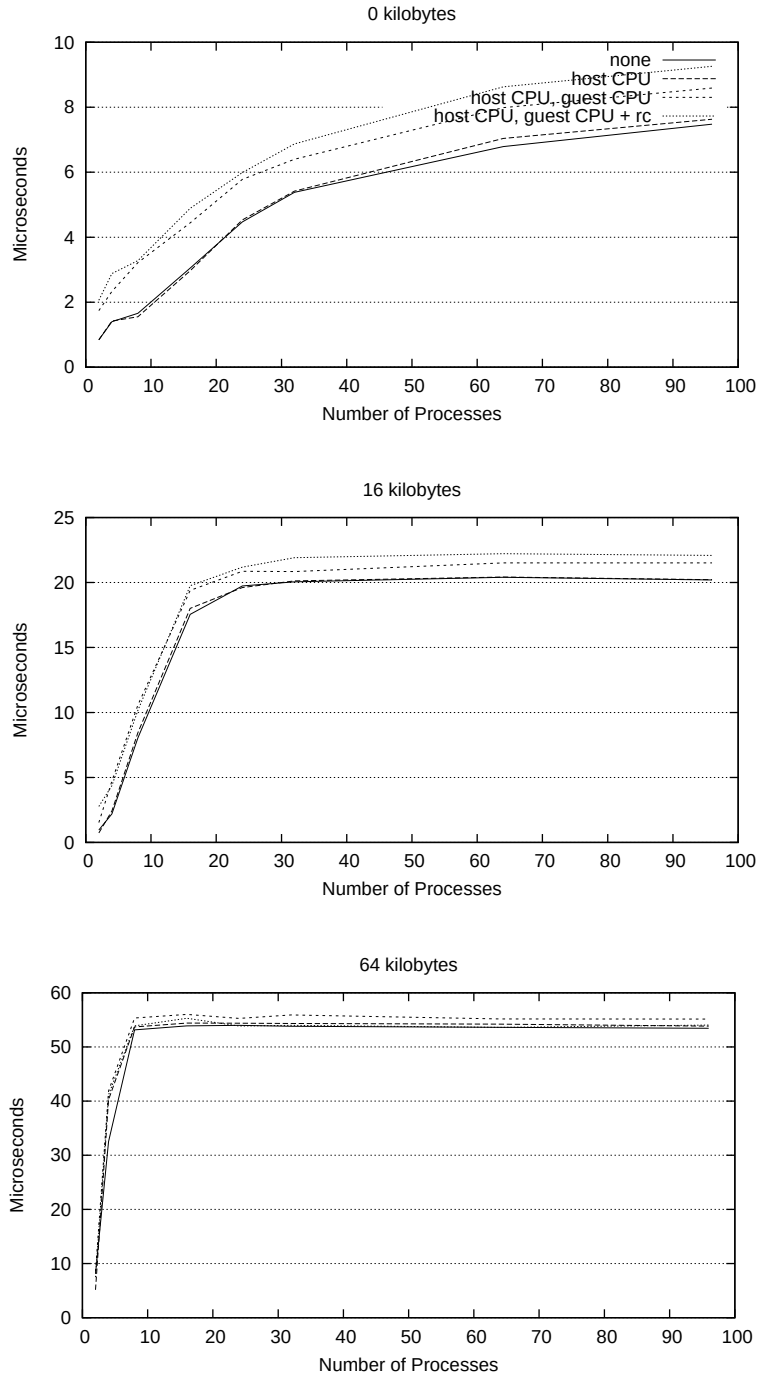


Figure 5.1: Context switch latency measured with lat_ctx.

"host CPU, guest CPU" At each context switch within the virtual machine, the guest operating system reads the performance monitoring counters and estimates the energy consumption of the previously running task. In addition, it analyzes the log file written by the microkernel to factor out the energy consumption of other virtual machines.

"host CPU, guest CPU + rc" In the last configuration, the guest operating system uses resource containers for accounting. It charges CPU energy consumption to resource containers instead of processes. In our case, each process is bound to a separate resource container. Thus, at each switch between processes, the kernel has to switch resource containers as well.

Figure 5.2.1 shows the context switching times for process sizes of 0, 16, and 64 kilobytes. As we expected, host-level CPU accounting does not affect the guest-internal context switches. The biggest part of the accounting overhead is caused by the guest-level energy estimation, which amounts to approximately 1 microsecond per context switch. Because the log file for virtualizing the performance monitoring counters is empty, we attribute this overhead to the reading of the counters from the processor.

Altogether, CPU energy accounting leads to a penalty of less than 2 microseconds for each context switch. In the unrealistic case of switching between two processes that do nothing, energy accounting doubles the cost per context switch. However, if we increase the number of processes and the process size, the relative overhead is getting smaller because the cache effects outweigh the processing cost.

5.2.2 Disk Throughput

In a second experiment, we measured the disk throughput with the Postmark benchmark. Postmark allows us to determine the performance of disk energy accounting and shows how the increased context switching latencies affect real applications. Postmark is an application-level benchmark that simulates the disk workload of a typical mail server. It creates a large number of files and performs several transactions on them. We configured Postmark to run 10000 transactions on 200 files with a file size ranging between 500 bytes and 1 megabyte.

Using a real disk, we could not observe performance penalties caused by our accounting infrastructure. Because of the large performance gap between current disks and the CPU, the cost for disk accesses outweighs the processing cost. In order to eliminate the overhead caused by the disk, we conducted our measurements on a ramdisk with a size of 256

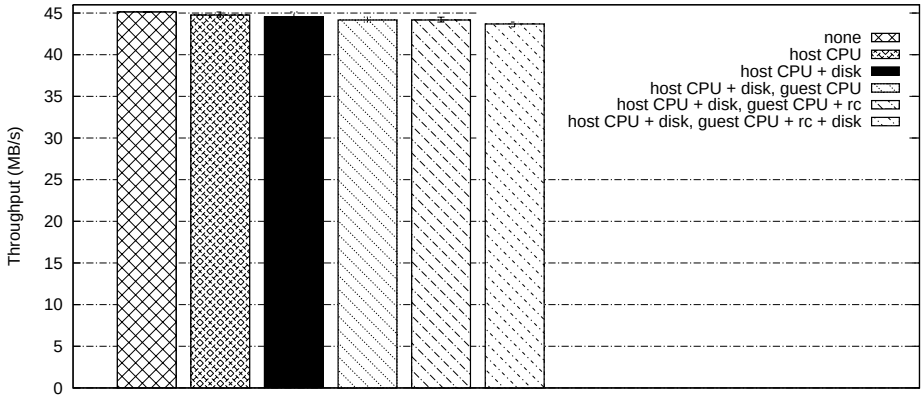


Figure 5.2: Postmark performance.

megabytes. The measured results are thus a worst case estimate for the overhead that is to be expected for real applications.

For the Postmark benchmark we used similar configurations as for lat_ctx but made some changes to measure the overhead caused by disk accounting. In particular, we introduced the configurations "host CPU + disk" and "host CPU + disk, guest CPU + rc + disk", which add disk energy accounting for host-level and guest-level. Accordingly, we changed subsequent configurations to include disk accounting as well. Hence, the configurations for the Postmark benchmark look as follows:

"none" See Section 5.2.1.

"host CPU" See Section 5.2.1.

"host CPU + disk" This configuration is identical to "host CPU", but with additional energy accounting within the user-level disk driver. For every request, the driver estimates the energy consumption of disk by means of an energy model[1]. Additionally, the driver estimates the CPU energy consumption every 50 requests and adjusts the cost for future requests.

"host CPU + disk, guest CPU" Identical to "host CPU, guest CPU" in Section 5.2.1, but with added host-level disk accounting.

"host CPU + disk, guest CPU + rc" Identical to "host CPU, guest CPU + rc" in Section 5.2.1, but with added host-level disk accounting.

"host CPU + disk, guest CPU + rc + disk" The last configuration contains our extension to the resource container implementation. The

[1] Although the benchmark operates on a ramdisk, we used the energy model of a real disk to get comparable processing cost for the energy estimation.

guest operating system uses resource containers to track disk requests through the page cache. It charges the energy consumption returned by the driver virtual machine to the resource container that is bound to the respective request.

For each configuration, we executed the Postmark benchmark five times and calculated the mean value. Figure 5.2 depicts the measured disk throughput for the six configurations. Except for the resource containers, each accounting component decreases the throughput between 0.5 and 1 percent, leading to an overall penalty of 3.2 percent. The results show that the cost can be attributed equally to CPU accounting and disk accounting, each adding 1.6 percent overhead. As mentioned before, the results are a worst case estimate for real applications; for typical server applications which use real I/O devices, the overhead will be even lower.

5.3 Accounting Information

In the remainder of this chapter, we will present energy measurements of several workloads to demonstrate the effectiveness of our approach. We conducted our energy measurements on a real IDE disk instead of the ramdisk used in the last section. However, to obtain realistic energy values for a server system, we estimated the disk energy consumption using the energy parameters for an IBM Ultrastar SCSI disk as reported by [4].

According to our two-level approach, we regard host-level and guest-level accounting separately. For both levels, we execute two instances of the same application simultaneously and observe how the accounting infrastructure apportions the energy consumption between the respective resource principals. In Section 5.3.1 we will regard host-level accounting and how it apportions energy consumption between virtual machines. In Section 5.3.2, we will focus on guest-level accounting.

5.3.1 Host-Level Energy Accounting

To demonstrate host-level energy accounting, we executed two instances of Postmark simultaneously, each in a separate virtual machine. During its measurements, the Postmark benchmark uses the CPU and the disk, allowing us to analyze the energy consumption of both supported devices. We started the second instance with a delay of ten seconds; together, the two runs take approximately 40 seconds. We obtained the energy consumption of the virtual machines from the resource monitor in a one second interval.

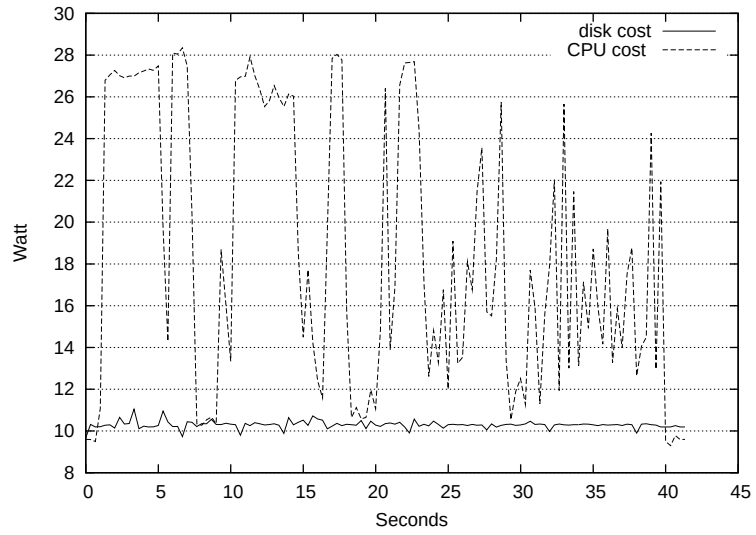


Figure 5.3: Total energy consumption of two simulaneous Postmark runs.

Figure 5.3 gives the total energy consumption of the two supported devices according to our estimation. As we can see, disk and CPU exhibit very different energy characteristics. The power consumption of the disk is dominated by the high base cost of approximately 10 Watt. Disk activity only slightly increases the power consumption, leaving little room for energy management policies without reducing the spinning speed. In contrast, the CPU energy consumption is very dynamic. Although the CPU base cost is lower that the disk base cost, its maximum power consumption is much higher.

In Figure 5.4 we solely considered the access cost during the experiment. We depicted three types of energy consumption separately: the disk energy consumption ("disk cost"), the CPU energy consumption that can be directly accounted to a client virtual machine ("direct CPU cost"), and the CPU energy consumption of the shared device driver ("processing overhead"). As we can see, the direct CPU energy consumption is by far the highest portion. The processing overhead in the device driver amounts to a total of 4 percent during the runtime of the experiment.

In Figure 5.5, we depicted the energy consumption of each virtual machine separately. As the chart shows, the resource monitor splits each part of the cost correctly between the virtual machines. In particular, the processing overhead in the device driver is accounted to individual virtual machines.

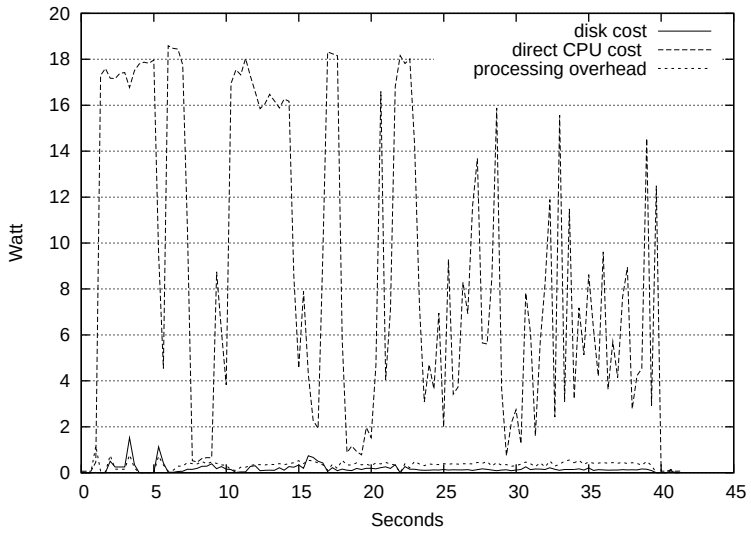
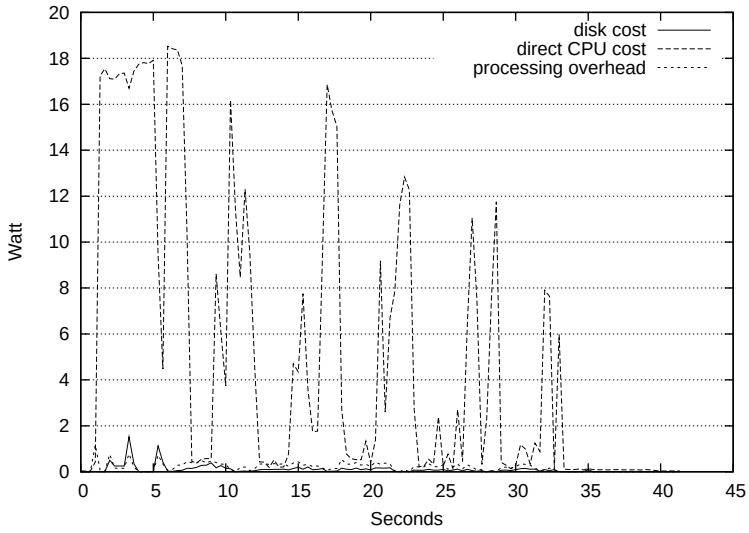


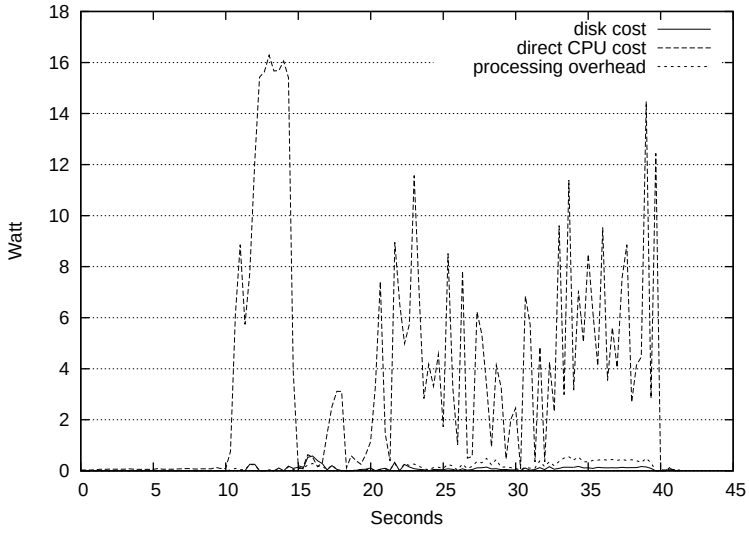
Figure 5.4: Access cost of two simulaneous Postmark runs.

5.3.2 Guest-Level Energy Accounting

In our last experiment, we executed two instances of *factor* within one virtual machine. The factor utility finds the prime factors of a number and is very CPU intensive. As with Postmark, we started the second instance of factor with a delay of ten seconds. Figure 5.6 shows the total energy consumption as seen by the resource monitor. Because the two factor instances run within one virtual machine, the resource monitor is unable to divide the cost between them. We used the accounting infrastructure in the guest operating system to obtain the energy consumption of the individual processes (see Figure 5.7).



(a)



(b)

Figure 5.5: Postmark access cost per virtual machine.

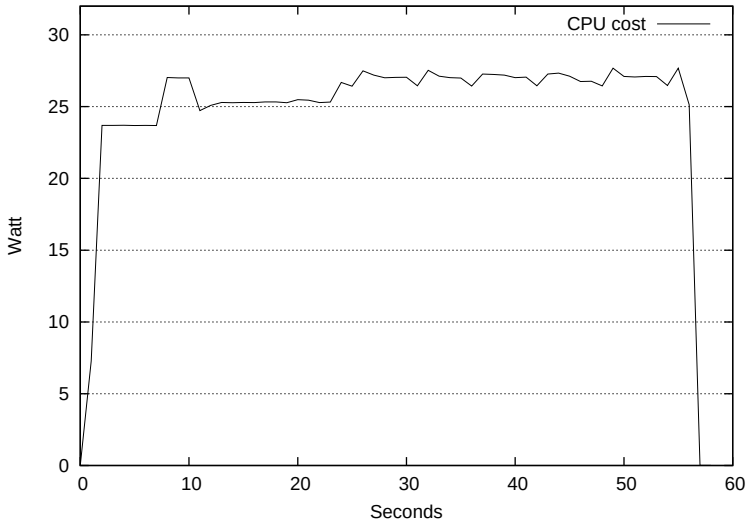


Figure 5.6: Total energy consumption of two simulaneous runs of factor.

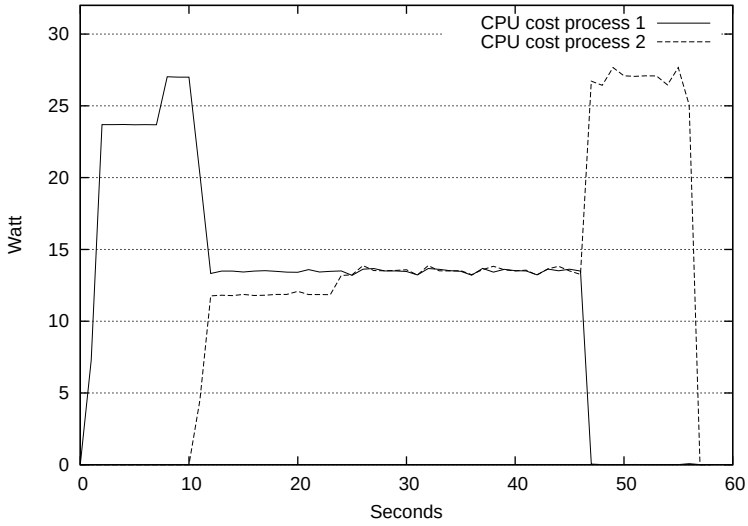


Figure 5.7: Energy consumption of each factor process.

Chapter 6

Conclusion

The goal of this thesis is to enable energy management in virtual machine environments by providing adquate accounting information. Although research has proposed several approaches to energy accounting, none of them is suitable for virtualized environments. All approaches are designed for monolithic operating systems and assume full knowledge of the hardware and the running applications. In virtual machine environments this information is distributed between the virtual machine monitor and the guest operating systems.

We presented an accounting infrastructure for hypervisor-based virtual machine monitors, that allows for reusing existing energy management approaches within the virtual machines. For this purpose, the virtual machine monitor estimates the energy consumption of the hardware and accounts it to the virtual devices. Depending on the virtualization technique applied by the virtual machine monitor, either the virtual devices simulate the energy characteristics of real hardware to support energy estimation within the guest operating system or we import the accounting information through paravirtualized device drivers. In addition, we provide accounting information to a resource monitor located at host-level, that serves as a place to implement global energy management policies.

Our work is a first step towards energy-aware virtualization. For active energy *management* in the virtual machine monitor, further work has to be done. Although we discussed in detail how to obtain the energy consumption of devices and of virtual machines, our solution does not address how the resource monitor can take appropriate actions to control it. This requires additional mechanisms and further extensions to the interfaces between the device subsystems and the resource monitor.

Besides mechanisms, energy management on host-level needs policies that identify global energy requirements and make appropriate decisions. In this thesis, we did not cover specific policies; however, we

see our infrastructure as a foundation for a wide range of energy management policies. Existing energy and thermal management schemes for server systems, such as [3] and [19], are promising approaches for virtualized environments as well. Futher work needs to investiate how they can be adapted to our infrastructure.

Bibliography

[1] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–58, New Orleans, LA, February 1999.

[2] Frank Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, Institut für Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, June 2001.

[3] 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)*, New Orleans, LA, September 2003.

[4] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, pages 86–97, San Francisco, CA, June 2003.

[5] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 387–390, Anaheim, CA, April 2005.

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

[7] Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[8] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–275, Seattle, WA, October 1996.

[9] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, San Francisco, CA, January 1994.

[10] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, Computer Laboratory, University of Cambridge, August 2004.

[11] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. DRPM: dynamic speed control for power management in server class disks. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, pages 169–181, New York, NY, June 2003.

[12] VMware Inc. Timekeeping in VMware virtual machines. Technical report, Palo Alto, CA, July 2005.

[13] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the USENIX 2003 Annual Technical Conference*, San Antonio, TX, June 2003.

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

[15] 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.

[16] Joshua LeVasseur, Volkmar Uhlig, Jan Stöß, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[17] Larry McVoy and Carl Staelin. LMbench - tools for performance analysis. `http://www.bitmover.com/lmbench/`.

[18] Andreas Merkel. Balancing power consumption in multiprocessor systems. Diploma thesis, Fakultät für Informatik, Universität Karlsruhe (TH), September 2005.

[19] Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-driven thermal management in SMP systems. In *Proceedings of the 2nd Workshop on Temperatur-Aware Computer Systems (TACS)*, Madison, WI, June 2005.

[20] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–167, Seattle, WA, October 1996.

[21] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accountable execution of untrusted programs. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, pages 136–141, Rio Rico, AZ, March 1999.

[22] Marcus Reinhardt. Cooperative, energy-aware scheduling of virtual machines. Study thesis, Fakultät für Informatik, Universität Karlsruhe (TH), August 2005.

[23] Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[24] Udo Steinberg. Quality-assuring scheduling in the Fiasco microkernel. Diploma thesis, Operating Systems Group, Technische Universität Dresden, March 2004.

[25] Jan Stöß. Using operating system instrumentation and event logging to support user-level multiprocessor schedulers. Diploma thesis, Fakultät für Informatik, Universität Karlsruhe (TH), March 2005.

[26] Adrian Tam, David Kar-Fai Tam, and Reza Azimi. Implementing resource containers in K42, November 2003.

[27] The K42 Team. Scheduling in K42, October 2001.

[28] The L4Ka Team. The L4Ka virtual machine environment. `http://l4ka.org/projects/virtualization/resourcemon/`.

[29] 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 2004.

[30] Martin Waitz. Accounting and control of power consumption in energy-aware operating systems. Diploma thesis, Institut für Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, January 2003.

[31] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, December 2002.

[32] Andrew Warfield and Keir Fraser. Now and Xen. *Linux Magazin*, October 2004.

[33] Andreas Weissel and Frank Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the First Workshop on Temperatur-Aware Computer Systems (TACS)*, München, Germany, June 2004.

[34] Heng Zeng, Carla Ellis, Alvin Lebeck, and Amin Vahdat. Currentcy: Unifying policies for resource management. Technical Report CS-2002-09, Department of Computer Science, Duke University, May 2002.

[35] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 123–132, October 2002.

[36] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 177–190, Brighton, UK, October 2005.