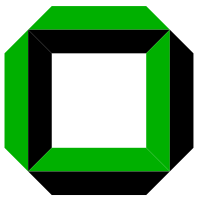

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Studienarbeit

Cooperative, Energy-Aware Scheduling Of Virtual Machines

Marcus Reinhardt

Verantwortlicher Betreuer: Prof.Dr. Frank Bellosa

Betreuender Mitarbeiter: Jan Stöß

Stand: 05.August 2005

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 08. August 2005 Marcus Reinhardt

Abstract

The concept of energy-aware scheduling offers new opportunities to react on energy consumption rather by the operating system than by appropriate hardware. Particularly systems can benefit from this concept, where typical hardware mechanisms like frequency scaling are not available or in environments where such mechanisms may not be applied. This work analyzes requirements for the adoption of this concept to a virtual machine environment. We presume that the virtual machines are cooperating by applying energy budgets assigned by a superior energy-aware instance.

Contents

Chapter 1

Introduction

The reasons for the energy-awareness of virtual machines are basically the same as for normal machines and have been addressed already in many papers, for example in [Wtz03].

Resulting from the lower CPU power dissipation the power required for cooling the CPU is reduced which allows server clusters being operated at lower cooling costs. A reduced operating temperature can also lead to a reduction of the size and the number of the required heat sinks and fans allowing to construct smoother machines. Furthermore, a malfunction of the CPU cooling device can be balanced.

However the mechanisms for energy-awareness may not be the same on virtual machine environments as on normal machine environments. In contrary to normal machine environments which run only one operating system in a virtual machine environment each virtual machine can run one operating system. These operating systems share all the physical resources. So they also share the CPU and its hardware mechanism for energy awareness like frequency scaling. But this mechanism and comparable approaches like halt-cycles are not designed to be shared. The virtual machines may be in different states requiring different handling of their energy related behaviour. Therefore such hardware based mechanisms can not be applied in a virtualization environment.

We elaborated a solution to this problem which gives the opportunity to achieve an energy-aware behaviour of virtual machines without applying any of the classic hardware mechanisms.

Our design based on scheduling tasks of virtual machines in an energy-aware context relocates the mechanisms for energy-awareness from hardware to software or rather into the operating system. A superior instance defines an energy budget which may be consumed by the virtual machines running. Therefore the energy budget is multiplexed to these virtual machines. It is their own charge to handle this budget in an appropriate way. We are going

to propose both a solution how to define the energy budget as well as we are going to show an approach how to handle the assigned energy within the virtual machines.

The implementation will cover the majority of the described components and mechanisms. We will discuss the extensions to virtual machines as well as those to the superior instance, consequently called hypervisor in order to make both energy-aware.

In chapter 2 we will first give an introduction to major contributions required for our work, namely virtualization, resource containers, energy approximation with performance counters and the Lm_sensors project. In chapter 3 the design will follow discussing the extensions to the virtual machines and to the hypervisor in an abstract manner. Chapter 4 will go into details of implementing the most important components of our proposed design notably the hypervisor and the virtual machines. Additionally energy behaviour of some minor components will be discussed. Chapter 5 will present some experimental results, for example measurements showing the preserving of a temperature limit with full CPU load. Finally chapter 6 will recapitulate our work and give an idea which direction future considerations may take.

Chapter 2

Background

The major contributions and their relations to our work will be presented in this chapter. Namely virtualization, resource containers and energy approximation based on performance counters are strongly required to implement our design.

2.1 Virtualization

Virtualization is a term used in different research areas with slightly different meanings. At first we want characterize what virtualization has been intended to be in the beginning in order to anticipate confusion. Though our design is not dependent on the used virtualization environment we want to give a short overview about the major facts of the virtualization environment we are going to use for implementing our design.

2.1.1 Virtualization In General

The general understanding of a virtual machine is described as an efficient, isolated duplicate of a real machine[Pop74]. The major part of the CPU instructions are executed in native mode. For others - called the sensitive instructions - direct execution has to be prevented. They need to be trapped because they may interfere with the state of the virtual machine and have to be replaced by routines provided by the interpreter, a part of the virtual machine architecture - one routine for each instruction trapped[Pop74].

2.1.2 L4Ka Virtualization Environment

The L4Ka Virtualization Environment offers the possibility to run multiple operating systems each in a virtual machine. In order to achieve this the physical resources are shared by the virtual machines. Therefore a further superior instance is required: the hypervisor. It multiplexes the physical resources between the virtual machines, a basic feature which will be instrumented within our design.

The L4-Linux clients are based on a Linux kernel ported to collaborate with the L4 Virtualization Architecture and may be run within a virtual machine.[Har97].

2.2 Energy-Aware Resource Management

As stated in the introduction hardware mechanisms like frequency scaling are not applicable to our problem. They are not shareable between multiple virtual machines. Therefore we need a mechanism supporting divisibility.

An approach supporting the demanded divisibility is energy-aware scheduling. Thus we need a mechanism for accounting and limiting resource consumption first. Than we need a metric representing the resource consumption.

Two major approaches exist for the representation of resource consumption in operating systems. Both approaches instrument resource containers to account their metric.

The first one represents resource consumption estimating the consumed energy by reading performance counters of a CPU[Bel03]. As we will use this later it is briefly summarized in chapter 2.5.

The second approach in [Zen02] introduces a new unit, the currentcy, which is the basis to characterize the power management of managed hardware resources. Thus it is required to allocate currentcy to each hardware resource and each task before distributing it to tasks.

2.2.1 Energy-Aware Resource Containers

The mechanism used for accounting resources consumption is an approach introduced in [Wtz03]. The author of this paper proposes an energy-accounting model built on resource containers compatible to old resource scheduling models, yet allowing new applications to use the full power of resource containers for the accounting. It is capable of obtaining a fair resource distribution by considering some special energy-aware cases. An example is a client/server situation accounting the energy consumption to the client triggering it rather than to the server which is just responding.

In addition, support for accounting different kind of resources is possible.

2.2.2 Resource Container Objects

Resource containers are kernel objects storing resource consumption of some part of the system together with some further information, like the resource limit or its maximum[Dru99]. These objects also include the remaining resource capacity for the attached task. The resource containers are organized in a hierarchical structure bringing some comfort when accounting the resource usage, for example between a task and its forked child.

2.2.3 Energy Approximation With Performance Counters

The selected metric for accounting resource consumption is energy because it is fairly distributable between the virtual machines.

That's why we need a way to estimate energy. A concept of energy approximation with performance counters is our choice. Though performance counters were created for performance tuning originally the authors of [Bel03] found a method to measure energy consumption this way.

Performance Counters

Performance counters have been introduced to the Intel architectures with the release of the Pentium PRO processor. They hold the task to count the occurrence of different performance events. These performance-monitoring events are intended to be used as guides for performance tuning[IntD3]. The performance-monitoring hardware consists of two main components: the event detectors and the event counters. By configuring them it is possible to obtain counts of a variety of performance events under different conditions[Sp02a].

Energy Approximation

The authors of [Bel03] introduce a concept for an efficient accounting model using performance counters to approximate the energy consumption. Furthermore an equation system transforming the desired CPU temperature limit into its corresponding energy limit is derived.

This concept will be applied to our design as the resource metric administrated by the resource container system which is summarized in section 2.2.

At first the equation to estimate the power limit of the CPU temperature

limit will be briefly introduced.

$$P_{limit} = \frac{-a_1}{2a_2} - \sqrt{\frac{1}{a_2}\left(\frac{T_{limit} - T}{c_2 dt} - a_0 + T + \frac{a_1^2}{4a_2}\right)}$$

where T_{limit} represents the desired CPU temperature limit and T the current CPU temperature.

The constant c_2 represents the characteristics of the interface material, heat spreader and heat sink in case of either rising or falling temperatures.

The constants a_0, a_1, a_2 can be found by measuring the static temperature T_s and the power consumption P. Afterwards an interpolation for the resulting points with a quadratic function has to be performed:

$$T_s(P) = a_2 P^2 + a_1 P + a_0$$

All constants have to be calibrated in order to adjust the equation for a specific system configuration.

Additionally [Bel03] describes the possibility to receive the consumed energy from a set of performance counters. By assessing them with their energy contribution per cycle we get a linear combination for the energy consumption based on performance counters (and the time stamp counter).

2.3 Lm_sensors

The Lm_sensors project is an approach to simplify the process of monitoring the hardware health of Linux systems offering the required health monitoring chips such as the LM78 or the LM75. The I2C bus (a more generic version of the SMBus) which is supporting the communication between the host controller and the hardware health chips is a further requirement for Lm_sensors to work. Only some of the chips are connected to the ISA bus and thus can be accessed by the common ISA-interface.

Their drivers provide the base software layer for utilities to acquire data on the environmental conditions of the hardware[LMS05]. Additionally the project brings along a library which allows access of the different hardware health chips for user programs. It abstracts from the underlying hardware and may also report other characteristic values like the CPU core voltage or the mainboard temperature besides the CPU temperature.

LM_sensors will be widely used as we need to measure the CPU temperature to calculate an energy budget.

Chapter 3

Design

The goal of our design is to schedule tasks of virtual machines in an energy-aware manner. Therefore we want to explain the usability of energy for our purpose. Afterwards our design will be introduced. At first the extensions to the hypervisor will be described. After that we are going to discuss virtual machine specific extensions. Then the energy consumption of both the driver (a part of the hypervisor) and the communication will be analysed.

Moreover you can find two algorithms to clarify the processes within the energy-aware hypervisor and within the virtual machines.

3.1 Virtualizing Energy

The missing divisibility is a basic reason why hardware mechanisms like frequency scaling can not be applied to a virtualization environment. Two different virtual machines may require two different proceedings at the same time, for example two different CPU frequencies.

Thus we need a mechanism for accounting and an accountable metric which is shareable between the virtual machines and therefore making each virtual machine energy-aware in an independent way. The mechanism is already known: the resource containers. But they can only be used if an applicable metric is found. An approach is described in [Bel03]. The authors demonstrate how to detect energy consumption by the operating system.

As energy is a metric which may be multiplexed to energy budgets for the different virtual machines, each virtual machine can schedule its tasks in an energy-aware manner independently from the other ones.

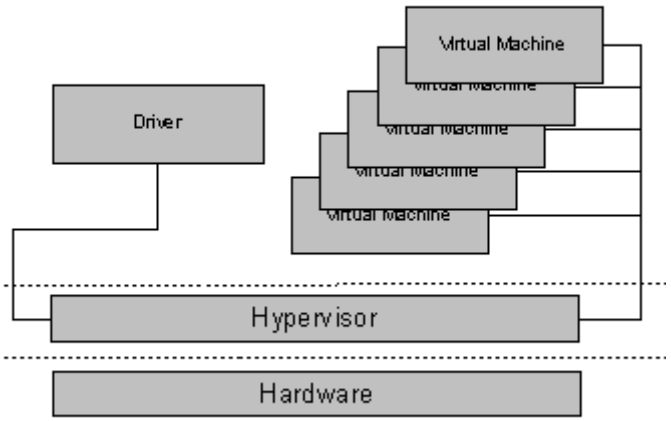

Figure 3.1: Basic configuration to schedule tasks of virtual machines in an energy-aware manner.

3.2 Basic Architecture

Our design enhances the components of an existing virtualization environment. The hypervisor is extended to be energy-aware and the virtual machines are adopted to react on an energy budget in a cooperative way.

As figure 3.1 shows the hypervisor basically needs to multiplex an energy budget between the virtual machines. This energy budget is calculated by a separate driver.

Accordingly the virtual machines have to assure that only the granted amount of energy is consumed by their tasks.

3.3 Energy-Aware Hypervisor

The hypervisor needs to determine an energy budget which may be distributed between multiple virtual machines.

Determining this energy budget is realized by a driver detecting continuously the energy consumption which is represented by an indicator concluding to energy in an appropriate way. In case of exceeding a predefined threshold this driver calculates a new energy budget. Afterwards the driver sends its results to the hypervisor.

Sending the results implicates communication between hypervisor and the

```
/* Maybe the driver is not available in the beginning */
Energy_Budget = ∞
n = getNumberOfActiveVMs()

loop

    Energy_Budget,New = updateEnergy()

    if Energy_Budget ≠ Energy_Budget,New then
        Energy_Budget = divideup(Energy_Budget,New, n)

        sendTriggerSignal()
    end if

end loop
```

Figure 3.2: Algorithm: The Energy-Aware Hypervisor

driver. Therefore, the hypervisor has to offer a suitable interface which will be called by the driver for this purpose.

Consecutively the hypervisor has to multiplex the energy to the different virtual machines. Therefore, the hypervisor needs to divide up the energy budget to a number of virtual machines.

Afterwards a trigger signal has to be sent to the virtual machines. In order to realize this we need to extend the interfaces of the hypervisor once again. The interface needs to be capable of informing the virtual machines about the new energy budget.

Figure 3.2 contains a summary of this chapter in an easy to remember algorithmic manner.

3.3.1 Driver Energy Consumption

The driver detecting the CPU temperature and eventually calculating the energy budget also consumes energy. This energy consumption will not be included by the mechanisms addressed above.

The driver calculates the new energy budget regularly on the basis of an energy indicator (the CPU temperature in our case). Prior to the calculation of the energy budget the indicator has to be detected as well. For this reason this energy amount will not be negligible low. But it can be considered static and therefore measured as it will perform the same task iteratively with a predictable frequency within a considered period. Consequently the energy budget can be decreased by this measured amount before sending it to the hypervisor.

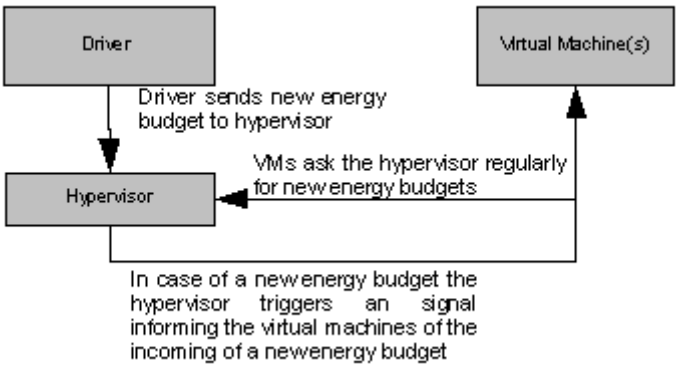

Figure 3.3: Communication between driver, hypervisor and virtual machines

3.3.2 Communication Energy Consumption

Figure 3.3 shows all communication between the three components driver, hypervisor and virtual machine. The whole communication must be analysed to see if it can result into significant energy consumption.

The energy consumption deriving from communication between hypervisor and driver can be predicted well, as it is known in advance how often communication between driver and hypervisor will occur. So it can be measured it for one communication process and afterwards be considered as static energy consumption in the same way like the driver energy consumption described in chapter 3.3.1.

Additionally the communication between hypervisor and the virtual machines has to be considered. If the energy consumption resulting from communication will not be negligible low, energy accounting needs to be interrupted before communication starts.

3.4 Energy-Aware Virtual Machines

The basic extension of the virtual machines is a mechanism preserving an energy budget. Therefore an energy accounting is required which supports multiple virtual machines running in parallel.

Figure 3.4 contains a summary of this chapter in an easy to remember algorithmic manner once again.

3.4.1 Energy Budget

At first the virtual machines have to be informed about an energy budget if a new budget has been calculated, ready to be fetched from the hypervisor. Therefore, the virtual machines have to call a suitable interface offered by the hypervisor. This happens either on a trigger signal or regularly based on the implemented algorithm.

3.4.2 Energy Accounting Within A Virtual Machine

A virtual machine needs to account the consumed energy to its origin. Consequently the energy consumption for each task running in the virtual machine has to be measured or rather to be estimated by an appropriate estimation function.

The context switch between two tasks can be instrumented to account the received energy consumption to the task which was running at last. Afterwards an appropriate mechanism has to state the energy at this point and then it has to track the energy consumption for further processing. A concept implementing this are the energy-aware resource containers found in [Wtz03].

Finally, the mechanism has to compare the energy emergence in its virtual machine to the energy budget received from the hypervisor. Based on this comparison a decision has to be made if new energy causing tasks may run or if they have to wait.

3.4.3 Energy Accounting Between Multiple Virtual Machines

A further problem has to be considered as there might be more than one virtual machine running: the problem of multiple accounting of energy.

If more than one virtual machine is running all these virtual machines will perform energy measuring. Thus energy measuring in the moment a virtual machine is not running needs to be interrupted. Therefore, the context switch between two virtual machines has to be instrumented to save the current state of the energy consumption. Later this state will be used as an offset for further energy measuring. Otherwise multiple measuring will occur and all virtual machines will account the consumed energy to the task they consider to be currently running. This results in a significant slow down of each virtual machine.

```
/* Maybe the energy budget is not available in the beginning */
Energy_Budget = ∞
Energy_Consumed = 0

loop

    if receivedTriggerSignal() = true then
        Energy_Budget = getNewEnergyBudget()
    endif

    if newAccountingPeriod() = true then
        Energy_Consumed = 0
    endif

    T = currentTask()

     /* Multiple accounting of energy has to be avoided in the next function */
    Energy_Consumed += getEnergyConsumption(T)

    if Energy_Consumed ≥ Energy_Budget then
        sendTaskToSleep(T)
    elseif
        scheduleTask(T)
    endif

end loop
```

Figure 3.4: Algorithm: The Energy-Aware Virtual Machine

3.4.4 Cooperative Virtual Machines

The mechanisms of keeping energy budgets are implemented in the virtual machines. Doing this the hypervisor has no possibility to enforce the correct behaviour of the mechanism. So the virtual machines are required to be cooperative.

A mislead virtual machine (for example an infected one by a virus) might not be cooperative anymore. If the hypervisor tolerates such a behaviour our design can be neutralized. To overcome this problem the hypervisor suspends virtual machines which turn out to be none cooperating.

Chapter 4

Implementation

A complete implementation of the design discussed above is beyond the scope of this work. But we have implemented the main components of the design to demonstrate its correct behaviour.

At first the extensions to the hypervisor realizing its energy-awareness will be presented. In doing so we will take a look at the driver, its connection to the hypervisor and its energy consumption, too. Thereafter the extensions to virtual machines with focus on its accounting mechanism will be discussed.

4.1 Configuration

The L4Ka Virtualization Architecture as described in chapter 2.1 is applied as virtual machine environment. It offers a hypervisor acting as a resource controlling instance and supports Linux 2.6 (L4 Linux) as client virtual machines.

Lm_sensors 2.9.1 introduced in chapter 2.3 is used to determine the hardware health characteristics, in this particular case the CPU temperature. This project supports a growing number of health chips and busses and a library abstracting from bus access and chip evaluation to receive the intended values.

The test machine runs with an Intel Pentium 4 at 1.5 GHZ without hyperthreading.

4.2 Energy-Aware Hypervisor

Three topics have to be discussed in this chapter. Firstly the virtualization of two instructions required by the virtual machines will be introduced later on. Secondly we are going to examine the driver. Thirdly the interfaces

offered by the hypervisor used for interaction between the components will be discussed.

4.2.1 Virtualizing Performance Counters

The virtual machines need to access two special instructions: wrmsr and rdpmc. They are introduced here as these instructions may only be accessed on hypervisor support. They are required to set up and read the performance counting registers.

wrmsr Support In Virtual Machines

The privileged instruction wrmsr (WRite to Model Specific Register) allows to write the model specific registers of a CPU. This includes the registers required to utilize the performance counters. This instruction has to be executed at privilege level 0. Such instructions may not be executed by the virtual machines directly as they are not running at this privilege level.

As they need to set up the performance events and counters the hypervisor has to provide access to it realized by a new interface.

The inferface

```
void msr_access( [in] pc_command_t c, [out] pc_result_t *r );
```

(implemented in hypervisor/msr_access.cc)

allows to send a command (pc_command_t) to be executed and returns either a 32-bit value or a 64-bit value encapsulated in pc_result_t, depending on the executed instruction.

User Mode Access To rdpmc

By calling the instruction rdpmc (ReaD Performance Monitoring Counters) the performance counters of a CPU can be read. This instruction can be set up by the hypervisor so it may be called from the user mode. That's why we do not have to port it to the hypervisor in contrary to the instruction wrmsr. Further instruction explanation will follow in chapter 4.3.3.

4.2.2 Extending The Hypervisor

The hypervisor is not energy-aware originally. To achieve energy-awareness first we have to extend the hypervisor by a driver capable to detect resource consumption and setting up the budget. Afterwards we have to assign the energy budget to the virtual machines. Therefore the new interfaces will be presented required for the three components driver, hypervisor and virtual

machines to be able to communicate.

The state of our implementation includes simplifications. We will only implement the mechanisms required to schedule one virtual machine. Thus instrumenting the context switch and multiplexing the energy budget between multiple virtual machines will be left out for future work.

4.2.3 Driver For Energy Consumption

The hypervisor needs information about the energy consumption and eventually the resulting energy budget. Both is done by the driver which we will discuss here.

Overview

For implementing the driver we will instrument Unmodified Device Driver Reuse. For further information concerning this topic see[JLV04].

As indicator for energy consumption we instrument the CPU temperature. Thus our driver continuously detects the CPU temperature and calculates a new energy budget if a preset temperature limit is exceeded. For the estimation of the energy budget from the CPU temperature we use an approach described in[Bel03].

This implementation contains a daemon called sensetempd. It is started up at the end of the boot process (note: this should mark the end of the complete boot process of all virtual machines; otherwise the virtual machines will boot up slowly).

It continuously detects the CPU temperature, calculates a new energy budget, wraps it to an event and finally sends it to the hypervisor.

The reading of the temperature sensors is done by using a library which is part of the Lm_sensors project. The code lines mentioned below are taken from sensetempd project file main.c

Structure

After initially checking the parameters sensetempd calls

```
set_machine( MACHINE_INTEL_P4_1P5 )
```

This command sets the machine to be used including event state calculation and threshold value selection (in this case to the test machine see chapter 5.1). The next crucial command is

```
LMS_init( configfile )
```

which initializes the Lm_sensors subsystem. By specifying a configfile it is possible to reduce the search space for probing the sensors on initialization consequently reducing considerable the time required to initialize and thus the energy consumed by the driver. Before entering the main loop the command

```
daemonize( PACKAGE )
```

makes sensetempd running as a Linux daemon.

Within the main loop first the L4 command

```
L4_Sleep( L4_TimePeriod(interval*1000) )
```

is invoked, ensuring the hypervisor is informed in the preset interval. The following piece of code exemplifies the temperature detection depending on the selected sensor.

```
if ( sensor==1 )
    // LMS_gettemp had be implemented to easily access sensors
    // by just passing chip and feature
    LMS_gettemp( &temp, LMS_getfeaturepos(1) );
if ( sensor==2 )
    LMS_gettemp( &temp, LMS_getfeaturepos(2) );
```

Now the state (the new energy budget) has to be calculated depending on the detected temperature. Afterwards it is sent to the hypervisor in case the state is a valid value (above or equal to one).

```
state = calculate_state( temp );

if ( state >= 1 )
    send_to_hypervisor( state, THERMAL_EVENT_CPU_TEMPERATURE );
```

The main loop runs until an error occurs or it is manually terminated by another sensetempd instance.

4.2.4 Event-Based Interaction

We need to report the energy budget from driver to the hypervisor. Afterwards the energy budget has to be sent to the virtual machines. Therefore events are sent between driver, hypervisor and virtual machines by using the hypervisor's interfaces.

A Container For Events

At first a new container object event_t has to be declared. It contains an event class (in order to support future extension) and the event state (in our case the new energy budget).

The event class is set to THERMAL_EVENT_CPU_TEMPERATURE, indicating a CPU temperature related event occurred and a new event state is available.

Furthermore there is an event class THERMAL_EVENT_RESET, notifying the hypervisor to reset all observed events. This is required if the driver needs to be rebooted. For this case the event state has to be set to maximum to ensure the virtual machines may still run.

Interfaces

The hypervisor offers 2 new interfaces defined in interfaces/hypervisor_idl.idl:

- int send_event([in] event_t e)
  The driver may submit an event as it occurs with this function to the hypervisor. The result may indicate an error (below zero) or a new interval length (above zero).

- int poll_event_state([in] event_t e)
  Virtual machines can check if an event occurred they are interested in. The result may report an error (below zero).

The interfaces are implemented in hypervisor/event_manager.cc and hypervisor/include/def_event_manager.h.

At Operation

As soon as the boot sequence has finished the daemon will continuously send the new energy budget. Firstly it will set up the event by setting the correct class and the new energy budget. Secondly it will use send_event(event e) to send it to the hypervisor.

Now the hypervisor is informed about the new event. It has to make sure the event state is available on the call of a virtual machine.

The virtual machines now may call poll_event_state(event) to get the latest event or rather their new energy budget.

4.2.5 Static Energy Consumption

As mentioned in the design energy consumed by the driver and the communication has to be analysed.

The driver is awakening frequently within an accounting period. As it is predictable how often this may happen the energy consumption can be measured for some cycles. As this results into a close to constant average value the energy consumption can be considered static. So the energy budget will be reduced by this value before sending it to the hypervisor.

Energy consumption occurring from interaction can be more complicated in case of multiple virtual machines running. As we excluded this case only interaction between driver, hypervisor and virtual machines will occur the frequency of which can be predicted. So we apply the same technique here as for the driver energy consumption.

4.3 Energy-Aware Virtual Machines

To support energy-awareness in virtual machines we have to extend them by four mechanisms. Firstly the virtual machines need to be extended with a mechanism for energy accounting. Secondly virtual machines need to be informed about the energy budget they may consume. Thirdly the energy consumed within the virtual machine needs to be estimated.

Fourthly each virtual machine has to be notified about energy consumed by other virtual machines in order to avoid multiple accounting. This last mechanism is left out for future work as we do not consider the case of multiple virtual machines.

4.3.1 Energy Accounting

We are instrumenting the resource container concept in[Wtz03], so we will discuss the required modifications only.

This resource container concept implements a task throttling mechanisms. Therefore no further modifications to the scheduler have been required. However we had to replace their metric (which is based on cycles) by our energy based metric. This was done in function res_cpu_account(void) in file include/linux/res_cpu.h. It is called in advance to a task switch so here the consumed energy will be stated and addressed to the corresponding task. As can be seen in figure 4.1 after it has been controlled that the idle task has not been running, the consumed energy for the period the task has been running is detected and addressed to the task. Afterwards the task switch is performed.

As the throttling mechanism and the extension to the scheduler is im-

```
void res_cpu_account(void)
{

    ...

    if (likely(current->pid))
    {
        energy = get_energy();

        rc_driver_use(RES_CPU, energy);
    }

    ...

    task_switch();
}
```

Figure 4.1: Replacing the old cycle based metric with the new energy metric

plemented within the resource accounting system it may be used with only one small modification. The used limit has to be replaced by our calculated energy budget.

4.3.2 Energy Budget And Events Processing

The hypervisor has been extended with an interface which may be called to receive the latest energy budget represented by an event.

The basic question here is where to attach the function to get the last event. The function call needs to be in advance to the refresh of the energy maximum as the received event state will directly interfere in this calculation. Furthermore it has to be close to this calculation so the applied event state is most likely the latest one. So an appropriate attachment position is in function void rc_driver_tick(void) in file driver/resource/rc_driver.c because there the refresh process is initiated. This position also ensures that updating the event state is not depending on the caller.

4.3.3 Energy Estimation

In this the concept of energy estimation derived from performance counters [Bel03] will be applied. Therefore in a first step the performance counters have to be set up, done by the function setup_perfcounter(). Now the appropriate counters can be read out. To read the performance counters the instruction rdpmc (ReaD Performance Monitoring Counters) has to be

```
unsigned int get_energy()
{

    ...

    // 0 is the number of the performance monitoring register
    // containing the readings for unhalted cycles.
    RDPMC( 0, buf64 );
    NEW_UC = (unsigned int) buf64;
        UC = NEW_UC - PMC_UC_OLD;
    PMC_UC_OLD = NEW_UC;

    ...

    // This formula is derived from[Bel03]
    energy = ( 6.17 * TSC ) +
             ( 7.12 * UC ) +
             ( 4.75 * MQW ) +
             ( 0.56 * RB ) +
             ( 340.46 * MB ) +
             ( 1.73 * MR  ) +
             ( 29.96 * MLR ) +
             ( 13.55 * LDM );

    return energy;
}
```

Figure 4.2: Example: reading the performance counters and estimating the energy consumption

called. The first parameter contains the number of the counter to be read (0-17, as the Pentium 4 has 18) and the second is a 64-bit variable to store the readings in.

After all counters have been read the consumed energy can be estimated which is implemented in function get_energy() as exemplified in figure 4.2. First the counter for unhalted cycles is read as an example how to read the performance monitoring counters. This is done for all events as described in chapter 2.3.3. Finally the energy is calculated and returned.

Chapter 5

Experimental Results

We want to present some experimental results to demonstrate the correct functionality of our design. Two experiments have been executed and evaluated for this purpose. The utility used within both experiments is the scp command included with the ssh package.

5.1 Preserving A Temperature Limit

The key point of our design is to schedule tasks energy aware in a virtualization environment. Therefore this experiment is going to show that our implementation preserves an energy budget which is derived from a temperature limit.

5.1.1 Configuration

Two scp instances running on different machines will copy an endless file to a single virtual machine. This results into a CPU load of 95% on the virtual machine which heats up the CPU to a maximum of 64°C if no throttling is available.

In the beginning the CPU temperature is at 41°C which is the lowest idle temperature which can be achieved. Our CPU temperature limit is set to 50°C.

5.1.2 Evaluation

Figure 5.1 demonstrates that the virtual machine preserves an energy budget derived from a temperature limit. The CPU's idle temperature of 41°C marks the entry point. The temperature begins to rise because of the high CPU load of 95% which is created by the two scp instances.

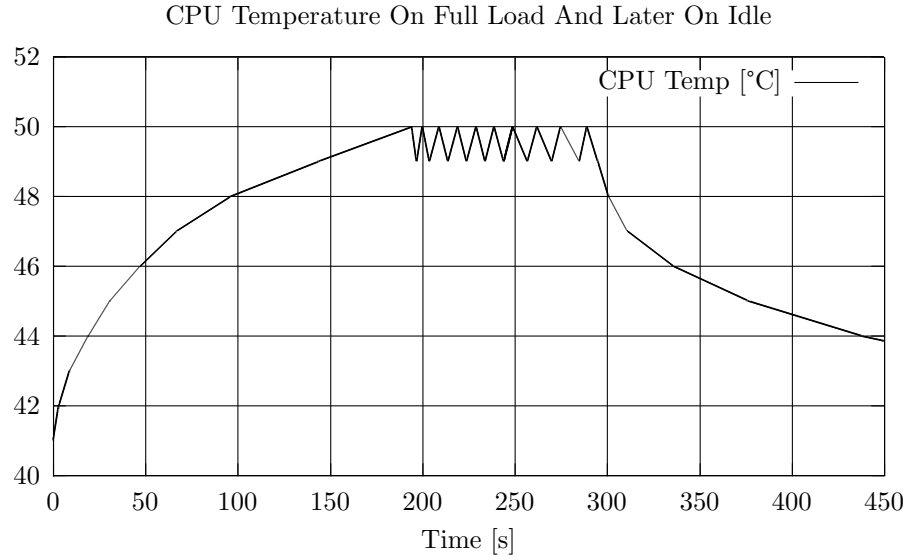

Figure 5.1: Three phases: warming up, preserving an energy budget and annealing

At 50°C the CPU temperature stops its rising and remains at 49°C and 50°C and thus preserves its limit. After longer test cycles the CPU temperature remains stable at 50°C.

After stopping both scp instances the CPU begins to anneal to the idle temperature.

5.2 Scaling Behaviour

The second experiment will demonstrate that our implementation scales in a predictable way. This means we want to achieve a comprehensible correlation between an internal representation of energy and an external representation. For example we want at enforced half CPU power only the half CPU load within a virtual machine. Furthermore we want to prove there is no impact of the number of applications running in the virtual machine to its maximum energy consumption.

Therefore we have to compare an internal representation to an external one representing an comprehensive and traceable metric.

5.2.1 Configuration

For internal representation we choose the power limit. For external representation we choose the transfer rates of the network communication caused

by multiple scp instances.

Firstly we will preserve a power limit at 30 watt, 45 watt and 60 watt to see if there is a correlation between internal and external representation. Secondly one, two and four scp instance will send a file to find out about impacts of different numbers of applications running.

Power Limit	SCP instances	Transfer rate for each instance	Sum
Watt	-	MBytes/Sec	MBytes/Sec
60	1	6.7	6.7
60	2	3.2, 3.4	6.6
60	4	2.0, 2.4, 1.1, 1.0	6.5
45	1	5.1	5.1
45	2	2.6, 2.3	4.9
45	4	1.0, 1.5, 0.8, 1.6	4.9
30	1	3.3	3.3
30	2	1.5, 1.8	3.3
30	4	1.0, 0.9, 0.6, 0.7	3.2

Figure 5.2: Transfer rates at 60, 45 and 30 watt for different numbers of scp instances

5.2.2 Evaluation

As shown in figure 5.2 the resulting transfer rates scale to 50% (for 30 watt) and to 75% (for 45 watt) compared to the results measured at 60 watt. Therefore we have a nearly linear correlation between energy consumption and network transfer rates.

Furthermore the sums of the transfer rates for the different numbers of scp instances are the same within each power limit. Thus we have shown there is no impact of the number of applications consuming energy to the maximum energy consumption.

Note: the fluctuation within a certain number of scp instances seems to be scp related as this occurs also on not virtualized Linux machines.

Chapter 6

Conclusion

We elaborated a design which enhances virtual machine environments with energy-awareness by scheduling tasks of virtual machines in an energy-aware context. Therefore our design considers all key components and contains two algorithms clarifying the processes within the components.

The implementation of our design covers most of the key points. Mechanisms for an energy-aware hypervisor have been introduced as well as those required for energy-aware virtual machines. We discussed the additional energy consumption of both the driver and the communication facilities and proposed a suitable and well applicable solution.

The experimental results demonstrated the operativeness of our design. It is capable of preserving a certain energy budget derived from the CPU temperature. Additionally external representations like the transfer rate of network communication scale nearly linear with the assigned energy budgets.

Future work can start with the correct information of energy consumption of virtual machines among each other. Furthermore, the distribution of the energy between more than one virtual machine needs to be explored, too. Another direction is to find a mapping of battery run-time to energy consumption in order to reach a desired system run-time.

Bibliography

[Uhl05] Rich Uhlig et al. Intel Virtualization Technology. IEEE Computer Magazin, May 2005.

[Pop74] Gerald J. Popek. Formal requirements for virtualizable third generation architectures. In Proceedings of the 4th Symposium on Operating System Principles, 1974.

[LMS05] Lm_Sensors Project Homepage. http://secure.netroedge.com/ lm78/, 2005.

[Wtz03] Martin Waitz. Erfassung und Regelung der Leistungsaufnahme in energiebewussten Betriebssystemen, 2003.

[IntD3] IA-32 Intel Architecture Software Developers Manual, Volume 3:System Programming Guide, 2004.

[Sp02a] Brinkley Sprunt. The Basics Of Performance Monitoring Hardware, IEEE Micro, July 2002.

[Sp02b] Brinkley Sprunt. Pentium 4 Performance Monitoring Hardware, IEEE Micro, July 2002.

[Mis03] Ramesh Mishra et al. Energy Aware Scheduling For Distributed Real-Time Systems, International Parallel and Distributed Processing Symposium, April 2003.

[Luo00] J.Luo and N.K.Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In Proceedings of International Conference on Computer Aided Design, November 2000.

[JLV04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gtz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04), December 6-8, 2004, San Francisco, CA

[Bel03] Frank Bellosa, Simon Kellner, Martin Waitz, Andreas Weissel. Event-Driven Energy Accounting for Dynamic Thermal Management, 2003.

[Mer05] Andreas Merkel, Frank Bellosa, Andreas Weissel. EventDriven Thermal Management in SMP Systems. Proceedings of the Second Workshop on Temperature-Aware Computer Systems (TACS-2), June 2005.

[Har97] Hermann Haertig et al. The Performance of -Kernel-Based Systems. Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP), St. Malo, France, October 1997

[Sto05] Jan Stoess. Using operating system instrumentation and event logging to support user-level multiprocessor schedulers, 2005.

[Zen02] Heng Zeng et al. ECOSystem: Managing Energy as a First Class Operating System Resource. October 2002.

[Dru99] Gaurav Banga, Peter Druschel And Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. 1999.