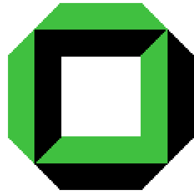

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Transparent, Thermal Balancing of Virtual Machines in Multicore Systems

Christoph Klee

Studienarbeit

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuende Mitarbeiter:	Dipl.-Inform. Jan Stöß

19. Oktober 2007

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 19. Oktober 2007

____
Christoph Klee

Abstract

Multicore architectures are employed more and more in todays system. A main task of current operating systems is to equalize the load of each CPU's core, it is accomplished by the migration policy of an operating system. Nevertheless, the migration policy of an operating system is mostly entangled in the kernel. Thus, migration policies of guest kernels running within virtual machines may contradict each other. Therefore, virtualization must break up applied policies from guest operating systems.

A virtual machine monitor allows to perform migrations independently of a guest. Thus, a virtual machine monitor can apply its own migration policy in order to fulfill system-wide requirements.

This thesis considers to load balance the power consumption of the cores of a CPU to decrease the overall emitted temperature as well as to reduce the system's power consumption. Therefore, our thermal balancing policy exchanges a virtual CPU with one consuming less power.

In order to determine a virtual CPU's power consumption, an energy accounting mechanism accounts each core's power consumption that can be assigned to a virtual CPU. To predict a virtual CPU's power consumption successfully for the next period, it is necessary that it does not change significantly. Thus, a core's power consumption is accounted and a virtual CPU is migrated very frequently.

Experimental results show that the overhead caused by migrating a virtual CPU up to a thousand times per second is acceptable but not negligible.

Contents

1 Introduction **1**

2 Background **3**
2.1 Power Management . . . 3
2.2 Migration . . . 3
2.3 Scheduling in L4 . . . 4
2.4 User-Controlled Scheduling on Top of L4 . . . 4
2.5 L4Ka Virtualization Environment . . . 5

3 Related Work **6**
3.1 Energy Accounting . . . 6
3.2 Energy-aware Load Balancing . . . 6

4 Design **8**
4.1 Architecture . . . 8
4.2 Problem Description & Analysis . . . 9
4.3 Virtual CPU . . . 10
4.4 Energy Accounting . . . 11
4.5 Energy-aware Virtual CPU Allocation . . . 11
4.6 Virtual CPU Migration . . . 15
4.6.1 Virtual CPU State . . . 16
4.6.2 Assigning Virtual to Physical CPUs . . . 16
4.6.3 Synchronization Path to Swap Virtual CPUs . . . 17
4.6.4 Resetting the Mapping Structure . . . 19

5 Implementation **20**
5.1 Afterburner Framework . . . 20
5.2 Afterburner Virtual CPU Model . . . 22
5.3 Accessing a Virtual CPU Object . . . 22
5.4 Guest User Threads . . . 23
5.4.1 Allocation . . . 23
5.4.2 Virtualization . . . 24
5.5 Migrating Virtual CPUs . . . 25
5.5.1 Migration During Guest Kernel Execution . . . 25
5.5.2 Migration During Guest User Execution . . . 27
5.6 Interrupt Handling . . . 29

6 Evaluation **31**
6.1 Network Performance 31
6.2 Kernel Build Performance 32
6.3 Physical CPU Dependent Object Accesses 33

Chapter 1

Introduction

Most of todays operating systems (OSes) are supporting the new multicore architectures, which have been launched in the last two years. To equally utilize a central processing unit's (CPU's) cores, threads have to be migrated between them by the OS. Although some OSes are supporting different scheduling policies, their migration policy is mostly entangled in their kernel. This prohibits to apply different migration policies.

Virtualization of a physical machine allows to run guest OSes isolated from each other on top of a small *virtual machine monitor* (VMM). Furthermore, it guarantees that the virtual machine itself and the underlying system or hardware are not effected of a guest system's crash. For this purpose, a VMM abstracts the hardware allowing a guest OS to run on top of one or more virtual CPUs. Thereby, more than one virtual CPU can be mapped to a physical one. Additionally, the physical CPUs can be shared among the virtual machines. This permits to achieve a better utilization of each core and balance their load.

Due to a VMM's introduced hardware abstraction, it is difficult for guest OSes to apply appropriate power management policies for their assigned resources, in particular for a virtual CPU. Additionally, the isolation of the guest systems causes that a guest system has merely a local system view. Especially, if guest OSes want to apply their policies to improve the system's performance or reduce their power consumption, the policies of the different guest OSes can contradict each other. This can result in an even worse performance than without their policies. As a consequence, virtualization has to break up the applied policies from guest OSes by applying an own VMM policy.

This thesis proposes a thermal balancing policy. It balances a CPU's power consumption among the cores of a CPU to reduce the system-wide power consumption and decrease the overall emitted temperature. Since a virtual CPU executing on top of a core causes mainly a core's power consumption and not the VMM itself, exchanging a virtual CPU with one consuming less power allows to distribute the power consumption among the cores.

Instead of migrating a thread that executes a virtual CPU from one core to another, our proposed migration mechanism merely exchanges the virtual CPU contexts of the involved threads. Since our thermal balancing policy exchanges the virtual CPU causing the highest power consumption with the one causing the lowest, only two threads are involved. The same applies for the remaining threads executing a virtual CPU.

In order to determine a virtual CPUs power consumption our policy demands an energy accounting mechanism that allows to assign a core's power consumption to a

virtual CPU. A virtual CPU's power consumption can change significantly during a few timeslices, because executed guest applications cause a changing power consumption. Nevertheless, our policy needs to predict a virtual CPU's power consumption to balance the power consumption among the cores. Therefore, a virtual CPU's power consumption is accounted more frequently than a guest OS schedules its threads. This permits to react quickly to a changing power consumption by an appropriate migration and to predict a virtual CPU's power consumption for the next accounting period more successfully. Furthermore, a short accounting period and frequent migrations reduce the impact of a mispredicted power consumption.

This thesis is structured as followed: chapter two gives a short review of power management, migration and the old and new scheduling approach in L4. In addition to that, it introduces the L4Ka virtualization environment. In chapter three, related work about energy accounting and energy-aware load balancing is presented. The design in chapter four considers our thermal balancing policy and its demanded energy accounting and migration mechanisms. Chapter five addresses implementation details to realize the design on top of L4. Finally, chapter six presents selected evaluation results.

Chapter 2

Background

This chapter gives a brief introduction of power management as well as migration. Afterwards, we present the old scheduling approach of L4. This has been replaced with a user-controlled scheduling approach. It allows to schedule threads without kernel interference. Finally, we explain the L4Ka virtualization environment. On top of this environment, we have implemented and evaluated the proposed design of chapter four.

2.1 Power Management

Today, power management of CPUs is becoming more and more important, this concerns not only users of mobile devices as laptops, but also computing centers. Users of mobile devices are expecting a long uptime of their devices without recharging batteries. Power management can increase this uptime especially by reducing the power consumption of its CPUs, e.g. by frequency and voltage scaling. Since a CPU may not reach a critical temperature, it has to be cooled down by an active air- or liquid-cooled heat sink to avoid its overheating. This heat sink is spending more power as higher the temperature of the cooled CPU is. By lessen the power consumption of the CPU, the power consumption of the heat sink can be reduced as well; in the best case also avoided.

Computing centers try to reduce their cooling costs by power management. If their applied power management policies achieve to reduce the power consumption of their servers, the hardware is less heated, they might scale down their computer room air conditioner and reduce the floor space for their hardware. Especially the cooling costs and the power consumed for cooling a computing center are becoming a major problem on the background of higher prices for power and emissions resulting from producing this power.

The next section discusses why migration is necessary and its impacts for power management.

2.2 Migration

Multi- and many-core architectures are becoming popular, they allow to run threads in parallel. To be able to run a thread on another core than its initial one, a thread has to be migrated from one core to another. The main purpose of migration is to keep the length of the runqueues of all cores more or less equal; this is called *load balancing*.

From the perspective of load balancing it will be important to migrate threads between different cores as infrequent as possible if each core has a distinct cache. A cache allows a core to access memory data with a lower latency than a memory access lasts. Normally, an application only requires to access a few memory data objects during a point of time; it is called an application's working set. If this working set is small-sized, it is possible to store it in a core's cache. The core to which an application is migrated to, may not contain the working set of the application. In that case, it is required to load these memory objects before they can be accessed. To ensure later on a low latency access they are stored in the core's cache.

Therefore, a thread would not be migrated from one core to another in a n core system, if only $n-1$ or less threads are runnable and each of these threads is running on its own core. Consequently, one or more cores are idle all the time. Power management can take advantage of this situation. By accepting to load a core's cache, a CPU's temperature and its power consumption can be decreased by migrating a running thread to a cold idle core. This allows to heat up cores more evenly. Nevertheless, it might also be useful to migrate load between two non-idle cores, if their workloads are resulting in different temperatures of a core.

2.3 Scheduling in L4

The older versions of the L4 μ-kernel have vectored out any interrupt, except for the timer interrupt. Therefore, L4 was responsible for scheduling its runnable threads and waking up threads, if they requested it. Since the scheduling decisions were made by the round-robin μ-kernel scheduler; they were intransparent for user-level threads.

Especially guest threads of a virtual machine and the performance of virtual machines suffered from this fixed scheduling policy. Because guest threads were scheduled independently of Linux by L4, threads were allowed to run, which had blocked in Linux, as long as they had not blocked in L4 as well. This resulted in a performance degradation of up to 86% as shown in [9].

In the next section we introduce our new user-controlled scheduling mechanism.

2.4 User-Controlled Scheduling on Top of L4

In order to prevent a performance degradation of a guest system due to an entangled scheduling policy in L4, it is necessary to permit user-level threads to schedule themselves. Therefore, a recent prototype of the L4 μ-kernel does not perform any scheduling anymore. To allow a user-level thread to schedule another thread, the timer interrupt must be handled by a scheduler as well. Since only one thread can be assigned as a timer interrupt handler per CPU, the root scheduler of a CPU handles timer interrupts. This root scheduler schedules the remaining user-threads that are assigned to its CPU.

A user-level scheduling implies that the kernel can not longer wake up threads, since it requires a scheduling decision by the kernel. This is contradicting a user-level scheduling approach, therefore one can only specify `zero` or `never` as a timeout for an interprocess communication (IPC). Given that the kernel does not longer dispatch threads, only one thread may be runnable per processor. The remaining threats are preempted and waiting for an IPC and time donation respectively.

2.5 L4Ka Virtualization Environment

Our design outlined in chapter four has been integrated into the L4Ka virtualization environment [1]. The environment is composed of four different modules: a hypervisor, a resource monitor, an in-place virtual machine monitor and a pre-virtualized guest OS [6]. The hypervisor is a recent prototype of the L4 μ-kernel, which supports SMP for x86. The resource monitor interacts with the μ-kernel to manage all HW resources as well as handling all interrupts, including the timer interrupt. Therefore, the L4Ka resource monitor is privileged to execute the privileged system calls of L4. The third module is the virtual machine monitor, which is called the afterburn wedge. It consists of two parts: an IA32 front-end and a L4 specific back-end. These two latter modules implement the virtualization services. The last module is a pre-virtualized Linux 2.6.9 kernel in which the afterburn wedge resides. This allows to execute most virtualization services in-place, avoiding expensive address space switches. Nevertheless, some services are requiring extended privileges, so that their in-place parts have to call their external parts, which are residing in the resource monitor, to fulfill them.

Per virtual processor, one afterburn wedge is running, thereby all wedges reside in the same address space. It consists of two threads, the monitor and a main thread. The main thread executes the code of the guest kernel and the monitor thread is responsible for resource management and scheduling the main thread.

Consequently, the L4Ka Virtualization Environment has a four level scheduling hierarchy. On top are the timer interrupt handlers, at the second level the monitor threads, followed by the main and guest kernel threads respectively and the guest user threads at the lowest level.

Chapter 3

Related Work

In the first half of this chapter, we present an energy accounting approach. It permits to predict the power consumption and temperature of a CPU by evaluating a CPU's performance counters. The second half considers different energy-aware load balancing policies, which try to reduce a CPU's temperature and increase its throughput.

3.1 Energy Accounting

One of the most important problems of today's microprocessors is their power consumption and temperature. To predict a CPU's power consumption as well as temperature *energy profiles* were introduced by Merkel & Bellosa [7]. They permit to estimate the power consumption of a task caused during one timeslice.

A task lifetime is divided in various phases of execution, which can be distinguished by varying power consumptions. Within such a phase the power consumption is more or less static, so that the last power consumption during the last timeslice can be used as a prediction for the next time the task becomes runnable again. Nevertheless, it is possible that the power consumption of a task changes significantly from one timeslice to another. This is due to a phase change of a task. How long such a phase lasts is defined by the input data, the actual executing algorithm and, additionally, by other running tasks, because they are causing unpredictable caching and paging effects.

Nonetheless, this phase changes occur rarely. As long as a substantial various power consumption lasts simply a few timeslices is not reflected by a noticeable temperature change because of a heat sink's thermal capacitance. It can merely be recognized by a changed power consumption.

To reflect not only the last timeslice, an energy profile can also account the energy spent during the timeslices before. Thus, short term changes of power consumption have not a big impact on a task's energy profile, whereas considerable long term changes are resulting in a changed energy profile.

At next, we discuss different energy-aware load balancing policies, whereas two policies are based on energy profiles.

3.2 Energy-aware Load Balancing

Observations by Powell, Gomaa & Vijaykumar [4] have revealed that the cooling time a core needs, after one of a core's essential resources has reached its critical temperature,

does not depend considerably on the number of resources that have to be cooled down. Therefore, the authors propose to heat up uniformly all resources of a core, to later cool them all together down. This approach is called *Heat-and-Run*. Nevertheless, one has to consider that a CPU's scheduling units are in use independent from the executed instruction, since every instruction has to be processed by a pipeline. Consequently, scheduling units are often the hottest ones on a chip [5].

In contrast to integer instructions that heat up merely integer units, floating point instructions equally use chip resources. To react in an appropriate way to this various resource utilization, it is important to monitor the power consumption not in a coarse grained manner. This means – in the field of multicore processors – not to measure only the complete processor but each core. Otherwise it is not be possible to throttle merely overheated cores, but one has to throttle all cores, which results in unnecessary throughput degradation.

In particular, each core of a Pentium D is provided with 18 performance counters, which can be used for monitoring each core's power consumption [3]. An accounted performance counter event is not directly related to its power consumption [2]. By multiplying a performance counter value with a weight, one obtains the energy consumption of the accounted event [3]. In addition to performance counters, temperature sensors are used by Lee & Skadron [5] for their *Heat-and-Run* approach. Their values allow to determine whether a core is overheated or not. If it is overheated, tasks will be migrated away from the overheated core to a colder one. Since migration is an expensive operation, it should happen as infrequently as possible.

The *hot task migration* is a similar approach [7]. It is not the aim to heat up a CPU to its critical temperature, but to migrate a task of a hot CPU to an evident cooler CPU, if the temperature difference between both has reached a defined threshold. To prevent a load imbalance the task of the cold CPU has to be migrated back. If the cold CPU is idle, a hot task can be migrated. In the case that the hot task could not be migrated, though, the CPU has to be throttled, which results in a throughput degradation. The characterization, whether a task is hot or cold, is determined by the energy profile explained in the previous section, which is only based on performance counter values. The same accounts for a CPU's temperature estimation. Its temperature estimation is based on a thermal model. Its input data are performance counter values.

The major drawback why a thermal diode is not even used for measuring whether the critical temperature of a CPU has been reached, is that reading a thermal diode implies significant overhead. It tasks 5.5 ms on a Pentium 4 [3].

To prevent throttling a CPU when applying the hot task migration, Merkel & Bellosa propose the *energy balancing* technique [7]. If a remote CPU's temperature is hotter than the local one and the tasks of the remote run-queue are exhausting more power than of the local one, tasks will be swapped. Cold tasks of the local run-queue will be exchanged with hot ones of the remote run-queue. The characterization which of the run-queues exhausts more power or is hotter, depends on the tasks' energy profiles belonging to the same run-queue. In the same manner as the energy profile, the estimation of a CPU's temperature is only based on performance counters. Tasks are exchanged between the run-queues to achieve an energy balancing without leading to a load imbalance.

In the next chapter we outline our thermal balancing policy to balance the cores' power consumptions. It is based on Merkel's & Bellosa's proposed energy balancing and hot task migration policies.

Chapter 4

Design

Todays operating systems migrate threads between distinct cores of a CPU to increase the system's throughput. They achieve this by keeping the length of the run-queues of the processors balanced. Balanced run-queues will guarantee an even turn-around time of each thread. This is ensured by a migration policy that is mostly entangled in an OS's kernel.

An entangled migration policy forbids a virtual machine monitor (VMM) to apply its own migration policy to fulfill system-wide requirements, since it cannot influence a guest's migration policy. Furthermore, a guest system has only a local system view, which forbids that a guest system A performs a successful thermal balancing migration policy, since a guest B's migration policy can contradict A's.

Therefore, a VMM must perform the migration on its own. Instead of migrating only a guest user thread from one virtual CPU (vCPU) to another one, it executes a vCPU on top of another physical CPU (pCPU). This allows exchanging a vCPU with one consuming less power to apply a thermal balancing policy. Its purpose is to reduce the system-wide power consumption and decrease the overall emitted temperature.

At first, this chapter introduces the structure of the architecture required for our proposed thermal balancing policy. Then we discuss, why it is required to load balance the power consumptions of the cores of a CPU. A core's power consumption is caused by a vCPU that is introduced afterwards. Applying our policy requires an energy accounting mechanism. This mechanism is outlined, before the thermal balancing policy is considered. At last, we introduce our migration mechanism to migrate a vCPU as required by the proposed policy.

4.1 Architecture

Virtual machines allow to execute guest operating systems in parallel. To multiplex the hardware between different guest OSes, a virtual machine has to abstract the bare hardware. Obviously, the most important resource to be virtualized is a CPU.

A virtual machine monitor (VMM), as shown in Figure 4.1, consists of threads executing within distinct virtual CPU contexts. These threads can either execute VMM or guest system code.

For this design, we assume that per virtual CPU offered to a guest system a vCPU thread executes within a VMM. Although a vCPU thread executes VMM and guest system code, it is not allowed to execute privileged instructions at all. Therefore, the

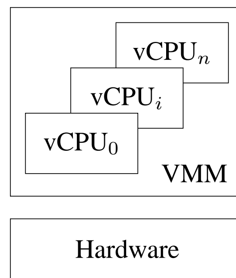

Figure 4.1: Structure of a virtual machine monitor

guest system must call the correlating VMM function of a privileged instruction; these calls are named *hypercalls*. Furthermore, a vCPU thread has to be able to notify another vCPU thread about outstanding events.

4.2 Problem Description & Analysis

A guest system tries to increase its throughput by migrating threads between its virtual CPUs without considering that a hot core needs to be throttled down. This throttling costs system performance and should therefore be avoided.

If a guest system considers not to overheat its virtual CPU, the core executing the virtual CPU's thread may overheat, because other guest systems also account for a core's power consumption and temperature. Due to the isolation of guest systems, a guest system can only consider its local contribution, but not the global contribution caused by all running guest systems.

A VMM does not suffer from a local system view like a guest system does. Therefore, it can prevent to overheat a core by an appropriate migration policy as long as not all cores of a system are overheated and hot respectively.

To balance the power consumption and temperatures of all cores of a CPU, a migration policy must know the temperature and power consumption of each core to make right migration decisions. Since a core's temperature changes too slowly and the resolution of a thermal diode is too low, an estimation of a core's power consumption is required. This estimation can be made by using an energy model for a core. From the core's performance counters it allows to derive the core's power consumption and temperature.

The estimation allows a VMM's migration policy not only to account the power consumption of a core, but also of the vCPU currently running on top of it. Consequently, one can distinguish between hot and cold cores as well as between hot and cold vCPUs.

Since a vCPU executes different guest applications with different energy profiles, a vCPU's power consumption may change significantly during a few timeslices. This makes it impossible to predict successfully a vCPU's power consumption for the next time period. Therefore, it is necessary to account a vCPU's power consumption very frequently to be able to migrate a vCPU more often than a guest schedules its threads.

Achieving that all cores of a CPU consume approximately the same amount of energy demands to exchange hot with cold vCPUs. Thereby, the migration policy has not only to consider the vCPU's current power consumption, but also the core's

power consumption it is running on over a longer period of time. The core's power consumption is required, since it includes the contributions of the remaining guest systems. Without their contribution a VMM's migration policy has only a local view of its guest system.

In addition to the in 4.4 outlined energy accounting approach, a migration mechanism is required to exchange hot with cold vCPUs. This is accomplished by our proposed migration policy described in section 4.5. As explained before, the migration has to happen very frequently to assure that the predicted power consumption of a vCPU will also be caused by it in the most cases.

In contrast to the migration of a vCPU thread including its vCPU context to another core, our migration mechanism described in 4.6 dynamically exchanges the context of a vCPU. Thus, a vCPU thread can execute another vCPU. This requires that the threads exchanging their vCPU contexts cooperate with each other. For this purpose, they have to pass by a common synchronization path to exchange their vCPU contexts, since a context will only reflect the current guest system's state if its thread executes VMM code. Due to the fact that our policy swaps a hot with a cold vCPU, merely two vCPU threads need to interact to exchange their vCPUs.

Before we discuss our thermal balancing policy and its required energy accounting and migration mechanisms, we outline the demands for a vCPU migration.

4.3 Virtual CPU

On top of virtual CPUs a guest system executes. Each virtual CPU is a memory object abstracting a physical CPU. A virtual CPU is reflecting the execution state of the guest system thread currently running on top of it. The execution state of a vCPU can simply be an instruction pointer, but it can even be a complete register frame.

Per virtual CPU that is offered to a guest system one virtual CPU thread exists. A virtual CPU thread runs within the context of a virtual CPU. This allows a vCPU thread to execute VMM code as well as guest code. As long as a vCPU thread executes guest system code, the data of a virtual CPU object will be invalid, since the guest system's execution state will be synchronized at first if the vCPU thread executes VMM code. A guest system's execution state allows a guest to proceed within the context defined by the execution state.

Due to the fact that a virtual CPU thread is assigned to a specific physical CPU, a virtual CPU is mapped to a pCPU. Therefore, it is possible that a few of its data structures relate to the physical CPU it is running on, e.g., the physical CPU id. In case a vCPU contains pCPU dependent objects, it is not possible to migrate a virtual CPU to a different physical one without any changes. If a vCPU contains no pCPU dependent data, a virtual CPU migration will only require to replace a vCPU object and its associated context of a vCPU thread with another one, thus a vCPU thread will execute afterwards in a different virtual CPU context.

Since a vCPU may only be executed by one vCPU thread, a migration of a vCPU requires a re-migration of another one. Besides, the vCPU threads would execute concurrently within the same vCPU context. As a consequence, a vCPU thread must be responsible for exactly one virtual CPU over its lifetime, but the vCPU does not necessarily have to be the same.

To be able to perform a vCPU migration, a vCPU object has to be split up into two parts: a physical CPU dependent and an independent part. In this way it is possible to migrate only the physical CPU independent part of a virtual CPU.

4.4 Energy Accounting

Applying a thermal balancing policy requires assigning to a vCPU its power consumption and temperature. Accounting a vCPU's power consumption and temperature necessitates performance counters and a core's energy model.

By evaluating selected events that can be accounted for, a vCPU's power consumption can be estimated. To suggest an accounted event's power consumption, an energy model must weight a performance counter value. Each core's power consumption during the last accounting period is provided to the VMM's policy. To avoid that the policy has only a local system view, each core's power consumption that has been accounted during a longer period is provided additionally. It includes the contribution of the remaining guest systems.

Depending on the required performance counters for an energy model, it is possible to read the counters sequentially, instead of explicitly. If not all performance counters are required, it will be unnecessary overhead to read each performance counter. Instead, only the activated performance counters need to be read. They are activated by distinct control registers.

As long as each ith control register corresponds with the ith performance counter, reading the ith performance counter requires to set its control register at first. Thus, instead of activating the control register by an assembler instruction directly, it can be done by calling an activation function performing the instruction as well. Additionally, this function marks the performance counter within a structure as active. Hence, the energy accounting function merely reads each entry of the performance counter structure. Only if the entry indicates an activated performance counter, the performance counter will be read as well.

Since the performance counter values are provided to vCPU threads, it can be omitted to provide values of non activated performance counters. Therefore, the values of activated performance counters are saved one after another. They are updated as frequently as requested by the thermal balancing policy.

Instead of providing merely raw performance counter values, the energy consumptions of the counted events are saved. This applies for the values accounted over a longer period and the ones of the last VM's timeslice. To indicate to the VMM's policy that a new energy consumption has been accounted, a bit is set in the provided structure which can be cleared by a vCPU thread.

During the activation of a performance counter one can specify whether the performance counter should generate a performance monitor interrupt (PMI) whenever the performance counter overflows. The interrupt delivery can be avoided as long as it is guaranteed that an accounted event does not lead to an overflow within an accounting period.

Hence, our in the next section outlined thermal balancing policy can account each vCPU's power consumption.

4.5 Energy-aware Virtual CPU Allocation

Our thermal balancing policy assigns vCPUs to pCPUs depending on a vCPU's power consumption and temperature. Thereby, the vCPU causing the highest power consumption is swapped with the vCPU causing the lowest power consumption, preferably with an idle vCPU. The same accounts for the remaining vCPUs.

Merkel & Bellosa [7] have shown that swapping a cold with a hot task can decrease a core's temperature as well as increasing the system wide throughput. Their policy has the advantage that it knows which task will be scheduled next. This allows their policy to consider a task's energy profile for its migration decisions.

In contrast to Merkel's & Bellosa's approach, a policy in a virtual machine environment cannot know which task will be executed next. This is due to the fact that a virtual machine monitor at most knows which task has been executed at last if it is saved in the vCPU object. The question which task is executed next is left to a guest OS. Therefore, the VMM policy can generally only regard a vCPU's power consumption, but not assign the power consumption to a guest task.

Hence, a mispredicted energy consumption of a vCPU will contradict a policy. This can happen whenever the guest OS schedules a task with an opposite energy profile than predicted by a policy right after a new mapping between vCPUs and vCPU threads has been set up. Additionally, whenever a guest schedules more than one guest task during an accounting period, more than one guest task's activity has contributed to a vCPU's power consumption.

One way to lessen this effect is to migrate vCPUs more frequently than the guest OS typically schedules threads. In that case, a guest system will continue with the execution of a guest task if it does not to have to handle an interrupt. Otherwise if it schedules a thread with a different energy profile, merely at the beginning of a guest timeslice the underlying mapping can contradict the policy. During the remaining parts of a guest's timeslice, the policy's aims will be fulfilled. This allows to react quickly to changing power consumptions of a vCPU.

Nevertheless, accounting only a vCPU's power consumption has the advantage that it is unimportant whether the energy consumption results from the kernel or from the user activity of a task. In comparison with Merkel's & Bellosa's policy that only migrates user threads but not kernel threads, our policy migrates vCPUs including their kernel and user threads.

Nevertheless, it is obvious that a migration between a hot and cold vCPU is not reasonable if the overhead for a migration is greater than its gain. Then, vCPUs might be migrated with a negligible different power consumption. Therefore, a threshold has to be defined controlling that only vCPUs are swapped whose power consumptions differed more than k% during the last period.

A disadvantage of this threshold is that a vCPU may consume k% more power than another one over its lifetime, not only over one period. To avoid a slow overheating of a core, while the remaining ones are kept cold, one needs to define when the requirement of an equalized power consumption is overruling the threshold and vice versa.

In contrast to Merkel's & Bellosa's policy that accepts a temperature difference between the cores over a longer period of time, we do not accept this. Their policy will migrate tasks only if the temperatures of the cores differ more than defined by a threshold. This threshold results from the costs of one migration. We consider that the costs of one migration can be amortized over more than one accounting period. Nevertheless, our policy is based on Merkel's & Bellosa's policy.

Since the main purpose of our policy is to minimize the energy gap between two cores, our policy assures – except for two cases – that the gap remains at least static or can even be reduced. The first case allowing the gap doubling at most is caused by the threshold, the second case allowing it is caused by a core's power consumption over a longer period. In order to prevent that a core heats up too fast, we accept an energy gap doubling at most. But only if an unchanged mapping causes a core that has consumed less power to consume more power than the other core after the next accounting period.

Nevertheless, this is only true as long as the cores consume the predicted power.

In order to discuss our policy in detail, we have to consider seven cases:

Let $E_i :=$ *energy, core$_i$ has consumed during c timeslices* and
$e_i :=$ *energy* > 0, *core$_i$ has consumed during the last timeslice* and
$k :=$ *threshold* > 1 be.

1. If $e_i = e_j \wedge i \neq j$, a swapping does not change the power consumption of i's and j's core. Irrespectively whether E_i and E_j are equalized, a vCPU migration will be avoided.

2. If core$_i$ is currently exhausting significantly more power than core$_j$, but the energy gap between both cores is bigger, the energy gap cannot be closed after the next timeslice (4.1).

$$e_i > ke_j \quad \wedge \quad E_j - E_i > e_i - e_j > 0 \quad \wedge \quad i \neq j \tag{4.1}$$

$$\iff \quad 0 < E_i < E_j + e_j - e_i \tag{4.2}$$

Therefore, the core's correlating vCPUs must not be swapped, otherwise the gap would be increased (4.3).

$$\overset{\text{no swapping}}{\Rightarrow} \quad \begin{array}{l} E_i' := E_i + e_i \qquad \wedge \\ E_j' := E_j + e_j \end{array} \tag{4.3}$$

Thus, the energy gap between both cores will be at least about e_j after the next timeslice (4.5).

$$\overset{(4.2),(4.3)}{\Rightarrow} \quad \begin{array}{l} E_i' < E_j + e_j - e_i + e_i \\ = E_j + e_j = E_j' \end{array} \tag{4.4}$$

$$\iff \quad E_j' - E_i' > \underbrace{e_j}_{\overset{(4.1)}{\Rightarrow} < e_i - e_j} \tag{4.5}$$

3. If the energy gap between core$_i$ and core$_j$ is not as big as in case two, the gap between both cores will be bridges without swapping the asssigned vCPUs (4.7).

$$e_i > ke_j \quad \wedge \quad 0 \leq E_j - E_i \leq e_i - e_j \quad \wedge \quad i \neq j \tag{4.6}$$

$$\iff \quad 0 \leq E_j \leq E_i + e_i - e_j \tag{4.7}$$

Although it fulfills the requirement of an equalized power consumption, core$_i$ has consumed considerably more power than core$_j$ in the last timeslice and probably also during the previous timeslices. In the case that the power consumption of both cores has been static, both cores' vCPUs have not been swapped because of their significant unequalized power consumption in the past. Therefore, it was not critical to heat up core$_i$, but now it may become critical. To prevent this, the policy swaps the core's assigned vCPUs (4.8).

$$\overset{\text{swapping}}{\Rightarrow} \quad \begin{array}{l} E_i' := E_i + e_j \qquad \wedge \\ E_j' := E_j + e_i \end{array} \tag{4.8}$$

Consequently, the energy gap between both cores will not be bridged, instead it will be doubled in the worst case (4.10).

$$\overset{(4.7),(4.8)}{\Rightarrow} \quad \begin{aligned} &E_j{}' \leq E_i + e_i - e_j + e_i \\ &= E_i + 2e_i - e_j \\ &= E_i{}' + 2(e_i - e_j) \end{aligned} \tag{4.9}$$

$$\iff \quad E_j{}' - E_i{}' \leq 2(e_i - e_j) \tag{4.10}$$

4. If the current power consumption of core$_i$ as well as its power consumption in the last c timeslices are higher than core$_j$'s ones, the cores' assigned vCPUs will be swapped (4.11).

$$e_i > ke_j \quad \wedge \quad E_i > E_j \quad \wedge \quad i \neq j \tag{4.11}$$

Thus, it is mandatory to swap the cores' assigned vCPUs (4.12).

$$\overset{\text{swapping}}{\Rightarrow} \quad \begin{aligned} &E_i{}' := E_i + e_j \quad \wedge \\ &E_j{}' := E_j + e_i \end{aligned} \tag{4.12}$$

5. As the main aim of this policy is to achieve an equalized power consumption of all cores, a vCPU migration has to counteract the effect of a slowly and steady increasing energy gap between distinct cores.

   Therefore, if the gap between E_i and E_j has the same magnitude as the gap between an e_l and e_m that would cause a migration because of the threshold criterion, a migration will be accomplished (4.13)

$$e_j < e_i \leq ke_j \quad \wedge \quad E_i - E_j > (k-1)e_j \quad \wedge \quad i \neq j \tag{4.13}$$

$$\Rightarrow \quad E_j + (k-1)e_j < E_i \tag{4.14}$$

In order to prevent an increasing energy gap between both cores, their correlating vCPUs are swapped (4.15).

$$\overset{\text{swapping}}{\Rightarrow} \quad \begin{aligned} &E_i{}' := E_i + e_j \quad \wedge \\ &E_j{}' := E_j + e_i \end{aligned} \tag{4.15}$$

Since the energy gap between both cores accounted during the last timeslice is less than the energy gap between them accounted during the last c timeslices, core$_i$ will have consumed also more power than core$_j$ after the next timeslice (4.16).

$$\overset{(4.14),(4.15)}{\Rightarrow} \quad \begin{aligned} &E_j{}' = E_j + e_i \\ &\overset{(4.13)}{\leq} E_j + ke_j \\ &< E_i + e_j = E_i{}' \end{aligned} \tag{4.16}$$

6. If the energy gap between both cores does not exhibit the required magnitude as in the last case, a migration will not be accomplished (4.17).

$$e_j < e_i \leq ke_j \quad \wedge \quad E_i - E_j \leq (k-1)e_j \quad \wedge \quad i \neq j \tag{4.17}$$

   Therefore, their assigned vCPUs will not be exchanged (4.18).

$$\overset{\text{no swapping}}{\Rightarrow} \quad \begin{array}{l} {E_i}' := E_i + e_i \\ {E_j}' := E_j + e_j \end{array} \quad \wedge \tag{4.18}$$

   Nevertheless, after the next timeslice, the gap may become the needed magnitude (4.19), so that case five applies next (4.13).

$$\overset{(4.17),(4.18)}{\Rightarrow} \quad \begin{array}{l} {E_i}' - {E_j}' \\ \leq (k-1)e_j + e_i - e_j \\ = e_i + (k-2)e_j \\ \leq ke_j + (k-2)e_j \\ = 2(k-1)e_j \end{array} \tag{4.19}$$

7. If $e_j < e_i \leq ke_j \wedge i \neq j$, core_j has consumed more power than core_i and $2(E_j - E_i) \geq e_i - e_j$, the energy gap between both cores will be bridged. Otherwise if $2(E_j - E_i) < e_i - e_j$, the gap will be increased. However, it will not be increased as much as if the vCPUs are swapped. Consequently, a migration may not be enforced in both cases.

The presented migration decisions of our policy are realized by the virtual CPU migration mechanism outlined in the next section.

4.6 Virtual CPU Migration

Our migration policy tries to reduce a core's power consumption and temperature by exchanging hot with cold virtual CPUs. This requires a migration mechanism that dynamically assigns vCPUs to pCPUs. It considers swapping vCPU objects and their related contexts, but not migrating a vCPU thread to another pCPU.

The mechanism tries to accomplish assigning a vCPU to a vCPU thread as requested by the thermal balancing policy. Given that only a hot vCPU is swapped with a cold vCPU, merely two vCPU threads must cooperate to realize a policy's requested migration. Therefore, both threads have to execute VMM code to assure that their vCPU objects are valid.

Our thermal balancing policy tries to predict a vCPU's power consumption, therefore it needs to react quickly to a changing power consumption of a vCPU. Otherwise a vCPU's power consumption would be caused by several guest tasks with different energy profiles, so that a successful prediction of a vCPU's power consumption would become impossible. Therefore, our proposed migration mechanism must not increase the system's latency significantly and it has to be possible that it can be called very frequently.

Before we consider our migration mechanism and how frequently it is called, we outline how a core's power consumption can be assigned to a vCPU. But at first we discuss when a vCPU forbids its migration and how it can be indicated to the VMM's policy.

4.6.1 Virtual CPU State

Since our migration mechanism migrates only the vCPU object and its associated context from one vCPU thread to another, a vCPU object has to be valid if it is migrated. It is synchronized whenever a vCPU's guest requires to execute a hypercall. Therefore, vCPU threads exchanging their vCPUs among each other are not allowed to execute guest code during an ongoing migration.

We have concluded in 4.3 that a virtual CPU must be split up into two independent parts: a physical CPU dependent and an independent one. As long as only objects of the independent part are accessed, the vCPU can be migrated, but during the accesses of objects of the dependent part a migration is forbidden. Otherwise a vCPU could access another vCPU's pCPU dependent data.

To avoid that a guest system will be migrated while it accesses objects of the physical dependent part of a vCPU object, the virtual CPU object has been enhanced with a flag to indicate whether it is feasible to migrate the vCPU.

Before entering a code section that requires to access a pCPU dependent object, the flag has to be set and the old value of the flag must be saved, thus this value can be reassigned after the critical code section has been passed by. It is important not to reset the flag, because in this case the virtual CPU state would wrongly suggest that it is allowed to migrate a vCPU within nested pCPU dependent sections.

The next subsection outlines how to ensure that a core's power consumption can be assigned to a vCPU, although a policy's determined mapping between vCPUs and vCPU threads is not reflecting the real system state because of pCPU dependent accesses.

4.6.2 Assigning Virtual to Physical CPUs

The migration mechanism has to try to migrate a vCPU as defined by a mapping between vCPUs and vCPU threads, which is determined by our migration policy. The mapping depends on the vCPUs' and cores' power consumptions respectively.

Given that the mapping is required for a migration, it has to remain unchanged as long as the migration of vCPUs is in progress. Besides, it must be prevented that more than one vCPU thread wants to execute in the same vCPU context as motivated in 4.3.

Due to the fact that each vCPU thread is executed at least once per timer interrupt, the mapping can be determined on each timer tick, but merely once and not by each thread. Therefore, a counter is required to ensure that the mapping is only determined by the first passing thread. The counter is incremented atomically by each thread after it has passed by the migration mechanism. Only if the counter equals the number of vCPU threads, a new mapping may be set up.

This ensures that a mapping can only be changed after all vCPUs threads have passed by the migration function. Therefore, each vCPU thread has to execute this function exactly once. Thereby, it is unimportant whether the mapping requires a migration or not.

If a determined migration of a vCPU is not realized because of a vCPU forbidding its migration, the old mapping will not reflect the real system state. However, to set up a mapping it is necessary to know which vCPU is executed by which vCPU thread to assign a core's power consumption to a vCPU. Therefore, a second mapping of vCPUs to vCPU threads must be introduced. It is always reflecting the real system state, except for the short period of time during an ongoing migration. Whenever a vCPU is migrated from one vCPU thread to another, the mapping is updated.

This allows accounting the amount of consumed resources for each vCPU. Nevertheless, this can only be assured if a new mapping is determined and realized by the migration mechanism shortly after a core's power consumption has been accounted and provided to the VMM's migration policy. Otherwise the migration mechanism described in the next sections, can perform a migration directly before the consumed resources would be accounted, thus the migration policy believes that the vCPU actually executing on top of the pCPU would have caused it.

4.6.3 Synchronization Path to Swap Virtual CPUs

The migration policy proposed in section 4.5 sets up mappings requiring that two vCPU threads exchange their vCPU objects. Therefore, it is sufficient that merely two virtual CPU threads cooperate in order to swap their vCPU objects. The remaining vCPU threads of a virtual machine are not affected by a migration of these two vCPUs.

A migration is accomplished by a migration function exchanging the pCPU dependent objects of two involved vCPU objects of an ongoing migration. Furthermore, it updates the memory references of the vCPU objects, hence a vCPU thread continues with its execution in the migration partner's old vCPU context.

In order to allow a vCPU thread to indicate its migration partner that it should swap their vCPU objects, a notification mechanism is required. Thereby, it has to be assured that merely one of two vCPU threads trying to swap their vCPU objects with each other sends a notification. Otherwise both threads are waiting for the notification reply of the other one forever.

Therefore, the mapping structure is enhanced by a *synchronization level* flag. Its main purpose is to indicate whether a notification has already been sent to the partner of the vCPU migration. Additionally, it indicates at which point of the migration mechanism a vCPU thread is.

The highest level indicates that the thread has not sent a notification yet, the second highest level that a notification has been sent and the lowest level that the migration mechanism has been passed.

Since the flag is read by the partner vCPU thread at the beginning of the swapping mechanism to state whether the other thread has already sent a notification, a common lock must be acquired. The following accesses caused by the migration mechanism can be performed without acquiring a lock, because concurrent accesses can be precluded.

By setting the *safe thread* flag, a thread indicates its migration partner a feasible migration. Additional to the synchronization level flag it is saved in the mapping structure.

When the first thread sends the notification, the partner thread can receive this message at two distinct situations. In most cases, the notification is received because the thread is waiting for an event. In that case, the thread is calling the swapping mechanism to perform the vCPU migration. Furthermore it is possible that the partner thread has already called the swapping function and determines with the help of the synchronization level flag that it is not the first thread. Therefore, it has to receive the outstanding notification at first.

Afterwards, the second thread can try to migrate the vCPUs. Only if both vCPUs and their safe thread states respectively allow a migration, a migration is permitted. In all other cases, merely a notification is sent to the initiator of the migration allowing it to proceed with its execution.

To perform the migration, the thread must exchange the pCPU dependent parts of the vCPU objects. Afterwards the memory references have to be updated, so that –

whenever a vCPU object is requested – the new one will be returned.

Before both vCPU threads can acquire their vCPU object once again, the callee must send a notification to the caller. The notification allows the callee to continue executing. At last each vCPU thread has to update the mapping reflecting the real system state, so that a core's power consumption can be assigned to the causing vCPU.

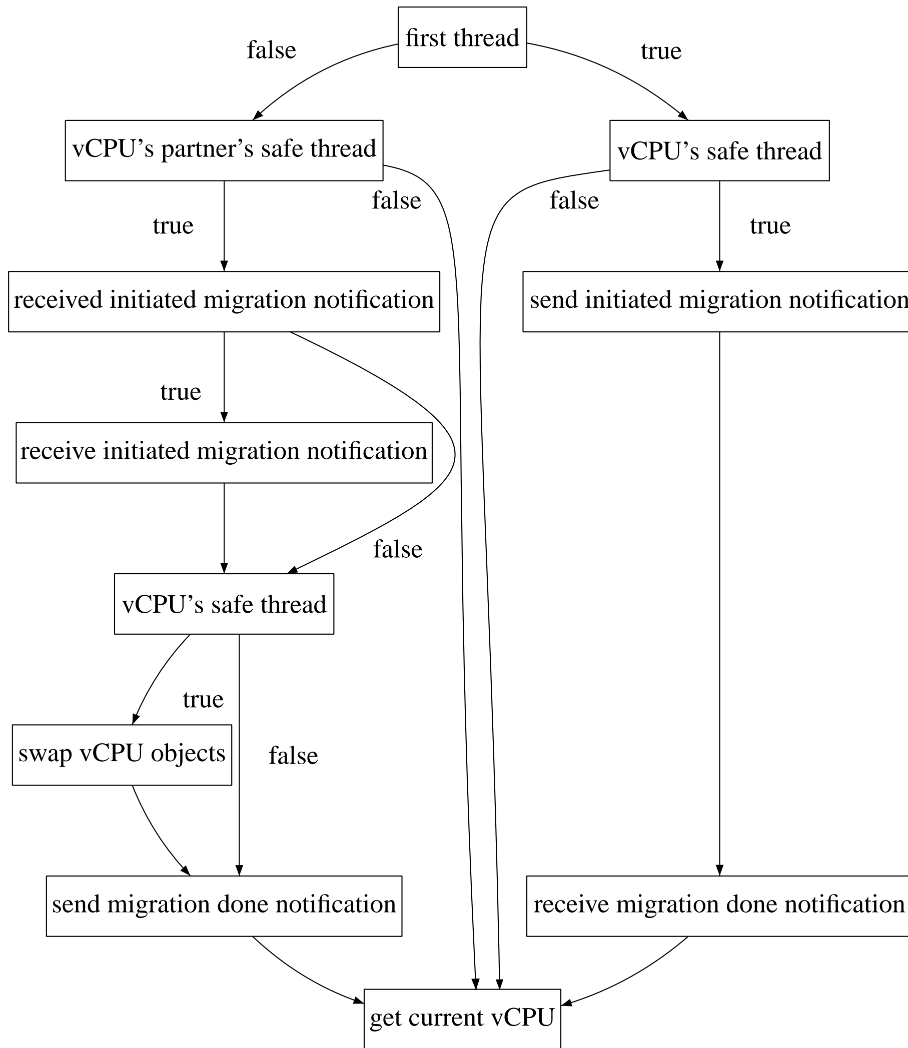

Figure 4.2: Flowchart to swap two vCPUs

Instead of calling the partner thread unconditionally whether a migration is feasible, it is sufficient to send a notification only if the vCPU state of the initiating thread allows a migration. If this is not the case, the synchronization level flag is set to "passed already the migration mechanism" and zero respectively, the safe thread flag is set to zero as well.

Accordingly, the second vCPU thread may not try to receive a notification from its partner thread. Therefore, the second thread has to check at first the other thread's

safe thread flag while it is holding a common lock. This assures that the flag is in a consistent state until the next new mapping. If it is indicating a failed migration, the thread must only set its synchronization level and safe thread flag to zero.

In the case that a vCPU should not be migrated to a different pCPU, the swapping method is simply passed by, its safe thread flag is ignored and its synchronization level flag reset to zero.

The operating sequence of the migration mechanism is shown in Figure 4.2. How frequently it has to be called and the resulting consequences are discussed next.

4.6.4 Resetting the Mapping Structure

A virtual CPU forbidding its migration prevents the realization of a determined mapping. To increase the probability that a vCPU is feasible to be migrated allowing to realize a vCPU migration, the migration mechanism can be called more frequently than a core's power consumption is accounted.

Additionally, due to asynchronous timer interrupts or system latencies it cannot be assured that a vCPU thread is calling the swapping function only once per new mapping. But as explained in 4.6.2, at the end of the swapping function, the counter indicating that a thread has passed this function, must be incremented exactly once by each thread per new determined mapping.

Therefore, the mapping structure must know whether a thread has already tried to migrate its vCPU and incremented the counter respectively. In this way, it can be ensured that each vCPU thread has passed at least once the swapping method before a new mapping structure has to be initialized. Only the mapping remains unchanged as long as the bit of the provided performance counter structure indicates that no new values have been accounted.

To enforce the new mapping as fast as possible the swapping function as well as the method setting up the new mapping has to be called before a vCPU thread returns from a hypercall.

Since a vCPU thread may execute the swapping method more than once per determined mapping, the mapping may change after a thread has passed the method at least once. Therefore, the mapping can change while a thread is executing within the function if the thread has already tried to migrate its vCPU. To assure that the mapping changes transparently for a vCPU thread, a thread must save – while it is holding a common lock – which vCPU it has to swap with as well as whether it has passed the method already. Afterwards, the thread is only allowed to read this copied data, so that its real status may have been changed in the meantime.

Thus, our thermal load balancing policy can migrate vCPUs very frequently to balance the power consumption of the cores and decrease the overall emitted temperature. In the next chapter we discuss details of the implementation of our design on top of L4.

Chapter 5

Implementation

This chapter addresses the realization of the proposed design on top of L4 and its integration into the afterburner framework [1].

At first, we present the afterburner framework and our afterburner virtual CPU model. It is compared with the virtual CPU model of our design. Afterwards, we discuss the problem of accessing a virtual CPU object. In the subsequent section we consider the allocation and virtualization of guest user threads required for their migration. The next section examines the implementation details of the vCPU migration including its guest kernel and user threads migrations on top of L4. At the end of the chapter we discuss the impact of the vCPU migration for handling interrupts and for the interrupt latency.

5.1 Afterburner Framework

For evaluating our thermal balancing policy and the performance of our proposed migration mechanism, we have integrated it into our afterburner framework.

The afterburner framework as shown in Figure 5.1 consists of four modules. The first module, a recent prototype of the L4 μ-kernel acts as a virtual machine monitor's hypervisor. Since it abstracts the bare hardware, it offers a VMM a high-level application interface (API) for handling hardware (HW) resources.

Our second module is the L4Ka resource monitor. It is permitted to execute privileged system calls of L4. These are required for handling HW resources as well as interrupts, including the timer interrupt. For handling each core's interrupts, one virtual IRQ (vIRQ) thread per core resides within the resource monitor's address space. Additionally, a roottask thread is located in this address space. It is implementing virtualization services requiring extended privileges. These services are called by the afterburn wedge, the framework's third module.

This third module implements the remaining virtualization services of the VMM. If a service cannot be handled by the afterburn wedge itself, because it requires extended privileges, the resource monitor's corresponding service is requested. An afterburn wedge resides within the address space of the guest system's kernel, our last module. Thus, virtualization services requiring no extended privileges, can be executed in-place avoiding expensive address space switches.

One afterburn wedge exists per virtual CPU offered to a guest system. All wedges of a guest reside within the guest kernel's address space. An afterburn wedge consists

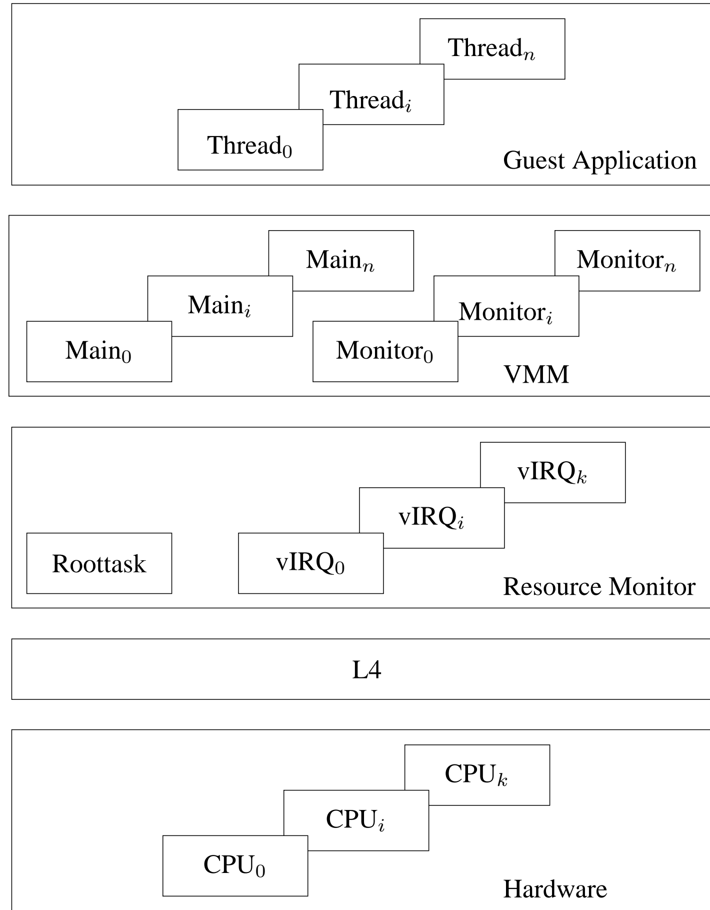

Figure 5.1: Structure of the afterburner framework

of a monitor and main thread. A monitor thread receives IPCs from its scheduler and vIRQ thread respectively, whenever an interrupt has to be handled. These interrupts are delivered to a guest kernel where an interrupt specific method handles the delivered interrupt. These methods are called interrupt handlers.

A vIRQ thread allows a monitor thread to proceed with its execution by sending it an IPC caused by an interrupt. Consequently, a vIRQ thread is the scheduler of a monitor thread. Since a vIRQ thread is not scheduled by any other thread, it is merely activated to handle an interrupt, it is the root scheduler of a core. Hence, a vIRQ thread accounts the consumed power of its core and offers its core's power consumption to the scheduled VMMs.

In contrast to a monitor thread that executes merely VMM code, a main thread executes guest system code as well. It is scheduled by its assigned monitor thread. One main task of a main thread besides executing guest system code, is to schedule a guest's user tasks. Each guest user task is spanned by its own address space consisting of one thread per vCPU. Thus, each main thread schedules its own thread of a guest user task.

Next we discuss our afterburner vCPU and how to accomplish its synchronization.

Additionally, we consider what needs to be done for updating a thread's execution state.

5.2 Afterburner Virtual CPU Model

The thermal balancing policy proposed in the previous chapter requires to migrate virtual CPUs by exchanging a vCPU thread's context. Therefore, the vCPU of our design described in 4.3 will only contain the current guest thread's execution state if its vCPU thread executes VMM code. Thus, a vCPU's execution state belongs either to a guest kernel thread or to a guest user thread. In contrast to that, the afterburner vCPU will contain the execution state of its assigned guest kernel thread and of the last executed guest user thread on top of its vCPU if the vCPU is synchronized.

Due to the fact that guest code is executed by a monitor's main thread, a monitor needs to update the execution state of its main thread permitting it to proceed within a new context. Therefore, a vCPU's execution state does not only contain an instruction pointer, but it contains the complete register frame of the main thread. The same accounts for the execution states of the guest user threads, since they have to be exchanged as well if a vCPU is migrated.

A thread's execution state must be synchronized with its assigned vCPU object whenever it gets preempted. For this purpose, a preemption message is generated by the μ-kernel and sent to the thread's scheduler. This preemption message contains the execution state of the preempted thread.

In the case that the scheduler has also been preempted and is waiting for an IPC, it receives the preemption message implicitly, otherwise it must receive the preemption message to synchronize the vCPU explicitly. This applies for the monitor as well as for the main thread. Both must synchronize the execution state of their scheduled threads before the migration mechanism may be called.

By sending a new execution state along with a preemption reply message allowing a thread to proceed, a thread can resume its execution in another context. Since the vCPU thread of our design has been replaced by the vCPU's assigned monitor, main and guest user threads, the migration mechanism must update the execution state of these threads. Additionally, the monitor and main thread's memory reference of the vCPU object has to be updated.

A thread's execution state can also be sent explicitly by calling the system call `L4_ThreadSwitch` sending a preemption yield message to the thread's assigned scheduler. It will be called by a main thread in order to yield its processor control to its monitor thread if its vCPU is idle.

The implementation of our design on top of L4 demands that the initial relationship between vCPUs and pCPUs can be obtained, e.g., for determining whether a vCPU can handle an interrupt. Therefore, the physical CPU dependent object (*pCPU object*) of a vCPU contains the *initial vCPU id*. Hence, each vCPU has an own pCPU object, although more than one vCPU can be mapped to a pCPU.

In the next section we consider the impact of preemptions on threads accessing their vCPU objects.

5.3 Accessing a Virtual CPU Object

Monitor and guest kernel threads access virtual CPU objects and execute within the context of their assigned vCPU object respectively. Since our proposed mechanism

migrates a vCPU object without the knowledge of a guest kernel thread, it is necessary that a guest kernel thread executes within the context of the new object, afterwards. If this is not the case, two distinct kernel threads would execute within the same context.

Therefore, we have to consider how it can be assured that two migrated threads are accessing their assigned vCPU objects. If a guest kernel thread gets preempted while it does not access a vCPU object, the new vCPU object will be returned to the thread after the migration. A vCPU migration is forbidden, during the access of pCPU objects, therefore it is not causing any problem. The first critical situation in which a thread can be preempted is while it acquires its vCPU object.

If a vCPU object is not acquired atomically, a thread could get preempted while obtaining its vCPU object. Since the migration mechanism cannot recognize this, it migrates the vCPU including updating the address of the thread's vCPU object. Thus, the thread continuing with acquiring the vCPU object, does not acquire its vCPU object. Instead of acquiring the old one of its migration partner, it obtains the migration partner's new vCPU object. This vCPU object was its old one.

In order to prevent that both threads access the same vCPU object, the thread needs to acquire the vCPU object again to obtain a valid one. To determine whether the vCPU object is valid, a thread compares its own processor number with the physical CPU id of the pCPU object. If both numbers are equal the thread may proceed, otherwise it must try to receive a valid object once again.

In case that merely pCPU independent objects of a vCPU are accessed, the vCPU can be migrated, because for these accesses it is not required to obtain the vCPU once again. Hence, the address of the vCPU object is stored on the stack.

One drawback of this approach is that a vCPU thread will get an invalid vCPU object if two vCPUs that are mapped to the same pCPU are exchanged. Nevertheless, this case will never occur, since vCPUs mapped to the same pCPU will never be exchanged.

5.4 Guest User Threads

Our migration mechanism discussed in 4.6 exchanges merely the vCPU contexts of two involved vCPU threads. Since the structure of our afterburner VMM is more complex than the assumed structure of a VMM in our design, it is not sufficient to update the monitor and main thread's vCPU references. Instead, for our implementation we have to consider the allocation of a guest application as well as its migration.

Before we examine which data structures of a guest application must be virtualized for performing a migration, we discuss how to manage an efficient guest application allocation.

5.4.1 Allocation

A user application of a guest OS executes in its own address space. This guarantees that a crash of a guest application can neither effect the VMM nor the guest system. Therefore, a guest user task and its address space needs to be allocated by the VMM whenever the guest OS creates a new application. At least one guest thread of a task has to be allocated. This allows to schedule a task on top of its assigned vCPU.

In order to permit a guest OS migrating its applications from one vCPU to another, a thread of a guest task will be allocated instead of migrated if no vCPU assigned thread of the guest application exists. Thus, a guest task's thread cannot only be allocated when a new guest address space is created, but also during an application's execution.

This is a drawback when a vCPU is migrated but no thread of the guest user task executing afterwards is allocated. To avoid a thread's allocation during the migration mechanism, one thread per vCPU is allocated during a guest application's creation. Consequently, during the execution of a guest user task none of its threads will be allocated.

To avoid expensive cross processor IPCs, a guest task's thread is scheduled anytime by the same main thread . Therefore, its scheduler and pager must be set accordingly. Since vCPUs may be migrated while threads of a guest task are allocated, it is mandatory that each thread has a distinct scheduler and pager. Thus, it is prohibited to rely on vCPU's properties, instead pCPU objects have to be accessed directly without the indirection of a vCPU object, because they remain unchanged after their initialization. Their initialization is accomplished before the first guest user thread is allocated. This ensures that a guest user thread's scheduler and pager are valid main threads.

5.4.2 Virtualization

Applying our thermal balancing policy, demands a migration mechanism. Our mechanism does not only have to swap and update vCPU objects of the involved monitor and main threads, but also to migrate vCPU's assigned guest user threads. Analogous to a vCPU migration, only a guest user thread's execution state has to be updated. This avoids changing a guest user thread's scheduler and pager during its lifetime.

A *thread info* object contains the execution state of a guest user thread as well as two thread dependent objects: a thread's thread id and its state. The thread id is required for replying to a guest user thread by a guest user thread's main thread. A thread's state indicates whether a thread is merely preempted, a page fault or exception has occurred, or if it is waiting for its startup IPC. Therefore, for migrating a guest user thread its thread info object has to be split up into two parts, a thread dependent and independent part.

In the case that a thread has not received its startup IPC yet, it must receive it before it can execute. If a thread has received the startup IPC already, it is unimportant which message it receives as long as it is not a startup IPC, since a startup IPC may only be received once.

Therefore, a thread's state needs to reflect a thread's real state only until the thread has received its startup reply message. Afterwards, the state is no longer thread dependent, so that the state can be migrated. Hence, two thread states may exist: the first state depending on its thread assures that a thread receives its startup IPC, the second, independent state that afterwards the requested message will be sent unconditional of a thread's real state.

If a guest user thread sends a message to its main thread, the message will contain at least the thread's execution state. This message is saved in the message registers of the receiving thread and main thread respectively. To assure that the message will not get lost or attributed to another thread, the used message registers have to be saved in the thread's thread info object at first. Afterwards, it is feasible to migrate the guest user thread as well as the vCPU.

Therefore, it is no longer permitted to rely on the message registers of a main thread, since they may belong to a different guest user thread. Accordingly, it is not allowed to migrate a vCPU as soon as the message registers of a main thread have been loaded because of its reply to a guest user thread. Otherwise a guest thread may receive a wrong message, resulting in an unpredictable user application behavior.

5.5 Migrating Virtual CPUs

In our design we have not distinguished between running a guest kernel or a guest user thread on top of a vCPU. Due to the VMM structure of our afterburner framework we must distinguish between them, since a monitor thread schedules merely a guest kernel thread, but no guest user threads as well. They are scheduled by their assigned main threads.

As long as a main thread executes guest kernel or in-place VMM code, a monitor thread must merely receive a main thread's preemption message for synchronizing its vCPU object, since the execution state of the last executed guest user thread has been synchronized before. In the case that a main thread schedules a guest user thread, a guest user thread's preemption message has to be received in order to synchronize its execution states with the main thread's vCPU object, before vCPU can be migrated.

Consequently, migrating a vCPU while it executes guest user code is more complex. Therefore, we discuss this case after we have outlined the mechanism to migrate a vCPU executing guest kernel or VMM code.

5.5.1 Migration During Guest Kernel Execution

Our migration mechanism does not only allow to migrate guest user threads, but allows to migrate a complete vCPU including its guest kernel thread. As long as the vCPU executed guest kernel or in-place VMM code before its preemption, our migration mechanism must merely synchronize the main thread's execution state with its vCPU object to swap its vCPU. The migration will be feasible, if the main thread does not access a pCPU object required for accessing a privileged VMM service. However, before we consider implementation details of our migration function, it is outlined what needs to be done to call the swapping mechanism and when to call it.

Minimizing the time between setting up a mapping and its realization, requires to call the swapping mechanism as frequently as possible. Although a monitor's event loop – for receiving IPCs and scheduling its main thread – does not only receive preemption, preemption yield and preemption reply messages, these three message types are the most frequently received ones. In order to avoid that the swapping method can only be called by one of these handlers because of their different code sequences, these three handlers have to execute a common code sequence after calling the swapping mechanism. This code sequence is called *swap reply*; it is a combination of their handlers.

Before the preemption and preemption reply handler of a monitor thread reply to their main thread, they check whether an interrupt needs to be acknowledged and delivered. This can be omitted in the preemption yield handler, since a vCPU can only be idle if no interrupt is pending.

Swapping a vCPU within the preemption yield handler changes the situation a bit, since the migrated vCPU executing after a performed vCPU migration is not necessarily idle. Consequently, also the preemption yield handler must check after a migration whether an interrupt has to be acknowledged and delivered.

Therefore, this is done by the swap reply method as well. It checks for interrupts, delivers them and replies to its main thread as long as the vCPU is not idle. If the vCPU is idle, a monitor thread sends a preemption yield message to its scheduler. The preemption reply handler introduces no more complexity, since it has a common code base with the preemption handler. Therefore, the swap reply function does not need to distinguish between them.

After we have discussed what is required to call our swapping mechanism, we explain under which condition the swapping method must and may not be called.

If a main thread yields its processor control, the preemption yield handler must save the thread's execution state in its vCPU object before calling the migration mechanism. The same applies for the preemption handler receiving a main thread's preemption message. It must call the swapping method independently from the question whether a migration is feasible or not.

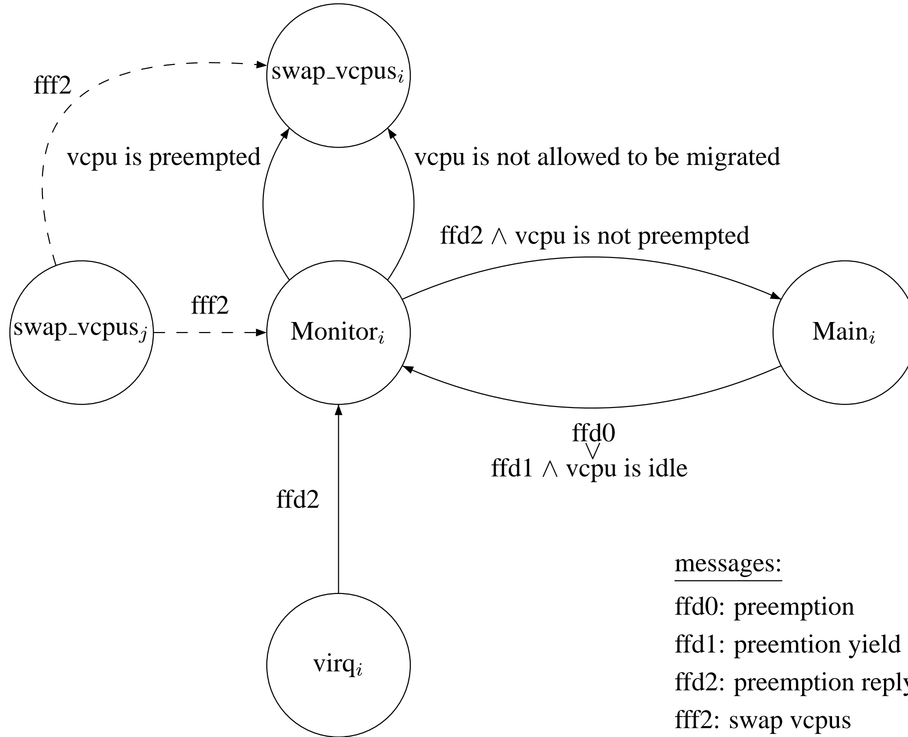

Figure 5.2: Swap a vCPU during guest kernel execution

In the case that a monitor receives a preemption reply message from its scheduler, a vCPU object may not be synchronized. If a vCPU is not synchronized, the monitor replies to `L4_nilthread` for receiving the outstanding preemption message of its main thread for synchronizing its vCPU object. Since this message is received by the monitor's event loop, the preemption handler calls the swapping function. This will not be necessary if a monitor's vCPU is already synchronized. Then, a preemption reply handler merely calls the swapping method. Furthermore, the swapping function can also be called if the vCPU state forbids its vCPU migration. Thereby, it is unimportant whether the vCPU is already synchronized or not, since a migration is not feasible because of its state and therefore must not be performed. This described flow is shown in Figure 5.2.

In subsection 4.6.3 we have outlined that a notification mechanism is required to swap two vCPUs. For our implementation on top of L4 we propose an IPC based notification mechanism, which is outlined next.

For swapping a monitor's vCPU, the monitor thread initiating a migration sends a swap vcpus message to its migration partner. If the receiving thread has already called the swapping method, the message must be received explicitly within the method. Otherwise this message will be received within a monitor's event loop by the swap vcpus handler.

Comparable to the preemption reply handler, the swap vcpus handler must receive the outstanding preemption message of its main thread if the vCPU is not preempted. If this is not the case, it has to call the swapping mechanism. Consequently, the preemption reply handler can also handle swap vcpus messages. It only has to distinguish, which message has been received to set the argument (*received swap vcpus message*) of the swapping mechanism appropriately to indicate whether the swapping method needs to receive the outstanding message or not. Furthermore, the preemption reply handler can avoid trying to update the mapping if it handles a swap vcpus message.

After a successful migration of a monitor's vCPU, a monitor must update – with the next reply to its main thread – its main thread's execution state by sending a preemption reply message. Therefore, a flag has to be saved in the vCPU object indicating whether a main thread's execution state must be updated. If the swap reply method replies to its main thread and the flag is set, it must update its main thread's execution state. Afterwards, it must reset the flag, in order to avoid updating the execution state even if it has not changed.

In this subsection we have discussed how to migrate a vCPU executing guest kernel or in-place VMM code, the next subsection addresses the migration of a vCPU executing guest user code.

5.5.2 Migration During Guest User Execution

The main task of a guest system is to execute guest user code. Therefore, it must be possible to migrate a vCPU executing guest user code and not only a vCPU executing guest kernel or in-place VMM code.

Due to the afterburner VMM structure, a main thread schedules a guest user thread by sending a guest user thread a preemption reply message. This requires to access the thread dependent part of a guest user thread's thread info object (5.4.2), therefore it is not feasible to migrate a vCPU while its main thread schedules a guest user thread. Furthermore, a monitor thread can merely receive its main thread's execution state, but not the execution state of the currently executed guest user thread required for a vCPU migration. A guest user thread's execution state can only be received by its main thread.

In order to allow a migration while a vCPU has executed guest user code, a monitor's main thread must be scheduled to receive the outstanding execution state of its scheduled guest user thread. It is saved in the thread info object of the user thread. Since, a main thread accesses no more thread dependent objects afterwards, it is permitted to perform a vCPU migration. For switching back to its monitor thread, a main thread performs `L4_ThreadSwitch`. This sends the thread's execution state along with the preemption yield message to its scheduler.

Since a main thread must only switch to its monitor thread if a migration is intended, the vCPU object has been extended by the flags *pending swap*, *ack swap* and *executing user*. A monitor thread sets the pending swap flag whenever a migration is outstanding and the main thread has set its status to executing user. A main thread sets this flag during the access of thread dependent objects for scheduling a guest user thread. After a main thread has received and saved a user thread's execution state, it checks whether a pending swap is outstanding. If this is the case, a main thread sets

the ack swap flag to indicate that a migration may be performed and switches to its monitor thread.

As long as the executing user and pending swap flags are set, a monitor thread knows that a migration will be feasible, since its main thread has to call the system call `L4_ThreadSwitch` within one timeslice. Therefore, any handler except the preemption yield handler must allow its assigned main thread to proceed for receiving the outstanding execution state of a main's guest user thread.

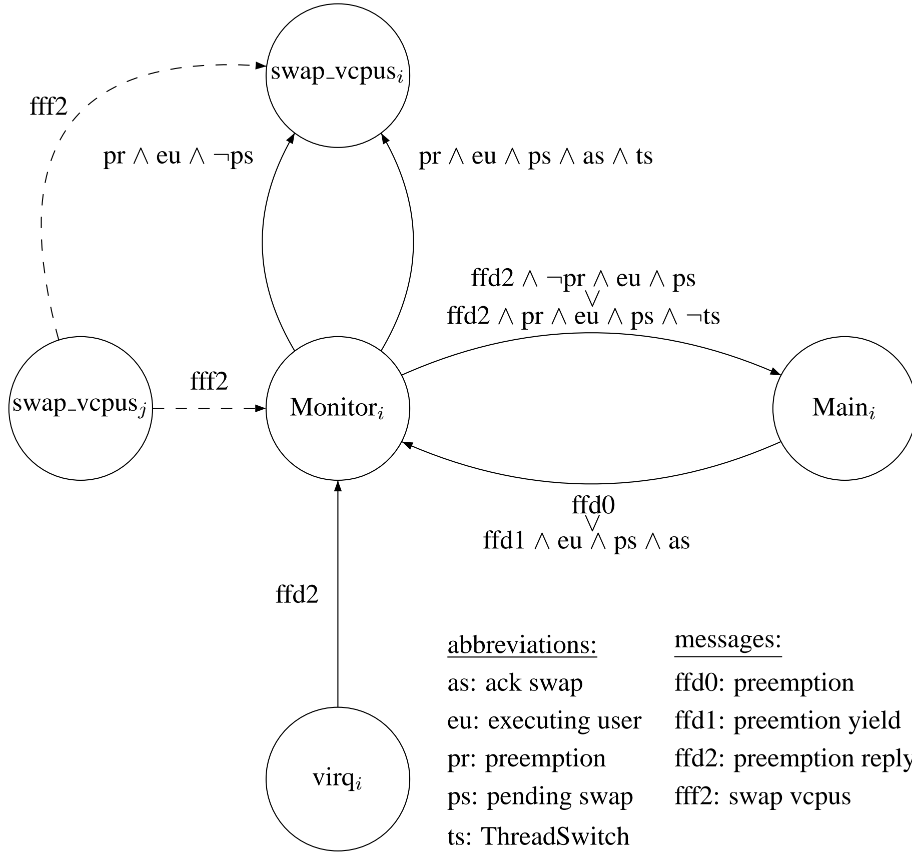

Figure 5.3: Swap a vCPU during guest user execution

If a main thread gets preempted before it calls `L4_ThreadSwitch` and has already acknowledged the pending swap, a vCPU migration must not be performed. Since its pending swap and ack swap flags are reset within the swapping method, a main's preemption yield message would be interpreted as an IPC, causing the monitor to send a yield IPC to its scheduler. Therefore, only the preemption yield handler is allowed to call the swapping function if a main thread has acknowledged a pending swap. This is shown in Figure 5.3.

A vCPU migration requires that a guest user thread's thread info object is set appropriately, as discussed in subsection 5.4.2. It has to be assured that a main thread replies only to its assigned guest user threads, but not to ones of other main threads. This can be assured by a main thread itself or in cooperation with its monitor thread.

Let us at first consider that a monitor cooperates with its main thread. During a successful vCPU migration each monitor thread has to update the current guest user thread's thread dependent part. This assures that a main thread accesses – within its event loop for receiving IPCs and scheduling its guest user threads – a valid thread info object required for replying to one of its assigned threads and especially for sending a startup reply message if a thread has not received one. Since only the object of the last executed guest user thread of a vCPU is adjusted by a monitor thread during the migration mechanism, the remaining objects reside unchanged. They must be adjusted on demand whenever a main thread schedules a new guest application.

In contrast to this first approach a main thread can assure without the assistance of its monitor to use a valid thread info object if it accesses the thread dependent part. Instead of adjusting the object only if a main thread schedules a new task, it is necessary to adjust it before each reply. By extending the thread dependent part of the thread info object with an *initial vCPU id*, it is possible to adjust the object and send the guest user thread a new execution state merely if the object's initial vCPU id differs with the vCPU's ones. In the case that both ids are equal, it can be avoided to send the new execution state along with a preemption reply message, as long as a main thread has received a preemption message. Otherwise the exception or page fault handler may have modified the thread's execution state requiring to send the new execution state.

Since the last approach will only send a new execution state if it is mandatory, it is called *lazy guest user thread migration*. In contrast to that, the cooperative migration is named *eager guest user thread migration*.

After we have discussed the necessary changes for migrating vCPUs in order to apply our thermal balancing approach, we propose a hardware interrupt notification mechanism in the next section. It reduces the impact of a vCPU migration on the interrupt latency.

5.6 Interrupt Handling

A vCPU migration caused by our thermal balancing approach inherently influences the interrupt handling of a virtual machine on top of L4. A vIRQ thread is assigned at least to its timer interrupt to handle a core's timer interrupt. Additionally, it can be assigned to hardware interrupts to handle them. Due to the afterburner structure, a vIRQ thread delivers an interrupt only to its assigned monitor threads.

Since the vCPU migration should be as transparent as possible, an interrupt must be delivered to its designated vCPU. One approach is that the migration mechanism migrates the IRQ threads of a vCPU as well. Furthermore, the vIRQ thread of a vCPU must be reassigned to its new IRQ threads to deliver their interrupts to its vCPU. The drawback of this approach is that an IRQ thread's migration as well as assigning a vIRQ thread to an IRQ thread requires to call privileged L4 system calls. In order to perform these calls two address space switches are required, since these calls can only be by accomplished by threads of the L4Ka resource monitor.

In order to avoid the overhead of these privileged system calls, we apply our second approach proposed next. It assures that interrupts are assigned to the same pCPU than without a vCPU's migration.

For this purpose, a VMM service must call a vCPU's interrupt handler that is not necessarily also its pCPU interrupt handler, whenever a device interrupt should be enabled or disabled. Although an interrupt is only delivered by a vIRQ handler to its monitor thread, each handler can handle every interrupt acknowledgement, even in-

terrupts not assigned to it. In order to acknowledge an interrupt a vIRQ thread is not assigned to, a thread must propagate the acknowledgement to an IRQ thread representing the interrupt.

Since the ongoing vCPU migrations are transparent for a vIRQ thread, a vIRQ thread delivers its hardware interrupts to its monitor thread. However, this is a problem, since a monitor does not execute in its initial vCPU's context all the time, but the hardware interrupt can only be handled and acknowledges by this vCPU. Hence, an interrupt can only be recognized if the vCPU's current monitor checks for them, which can last up to one timeslice.

This may become a bottleneck especially for hardware interrupt intensive applications. Thus, a hardware interrupt notification mechanism between vIRQ and their assigned monitor threads as well as between the monitor threads themselves is required.

The hardware interrupt notification between vIRQ and their monitor threads can be accomplished by setting a flag indicating an outstanding hardware interrupt. Before a monitor thread waits for an IPC from its scheduler or main thread, it checks the flag to distinguish whether it must notify another monitor thread about an outstanding interrupt.

Only if the monitor thread does not execute or has executed before the last tried migration in its initial vCPU's context, it must send the monitor executing in its initial vCPU's context a hardware interrupt notification. It enforces that this monitor handles the outstanding interrupt. Otherwise, it has handled the interrupt itself and no interrupt is outstanding anymore, or the interrupt will be handled by its migration partner. After sending the notification or handling the interrupt, the flag is reset by the monitor thread.

Our proposed interrupt notification mechanism is not required for timer interrupts, since each guest kernel thread can handle a timer interrupt. Thereby, it is unimportant whether a timer interrupt is designated for the monitor's current vCPU, since each monitor delivers a timer interrupt to its current vCPU.

In the next chapter we present selected evaluation results of the implementation of our design on top of L4. In particular, we consider the eager and lazy guest user thread migration as well as the benefit of our proposed interrupt notification mechanism.

Chapter 6

Evaluation

The performance of the implementation of our proposed design on top of L4 has been evaluated on a 3 GHz Pentium D830 with 2 cores and 2 GByte memory. A virtual machine consisting of one guest kernel thread per core has been executed on this system. The virtual machine's timeslice lasts 1 ms and the guest's timeslice 10 ms. A migration frequency of 0 Hz in the following benchmarks is equivalent to the afterburner performance without any changes.

Due to the lack of a energy model of the system, it has not been possible to apply the proposed energy-aware vCPU allocation policy. Instead, the applied policy tries to migrate the vCPUs as often as possible to achieve 1,000, 100, 20, 10 or one migration per second.

At the beginning of this chapter, we present the network performance of our implementation. Thereby, we compare the eager with the lazy guest user thread migration. Furthermore, we show the benefit of our proposed interrupt notification mechanism for the network performance and interrupt latency. Afterwards, we examine the overhead introduced by our migration mechanism by building the Linux kernel. We finish this chapter with an evaluation of the impact of physical CPU dependent object accesses on the realization of determined mappings by the migration policy.

6.1 Network Performance

For measuring the network performance and interrupt latency respectively, an Intel E1000 Gigabit network interface has been attached to the system. The I/O load has been generated by the `Netperf` benchmark [8], which has been executed by an external client.

As outlined in 5.5.2, a guest user thread can be migrated eagerly or lazily. These two approaches have been compared with each other, whereupon the migration policy tries to migrate a vCPU every 1, 10, 50, 100 and 1,000 ms.

As you can see in Table 6.1, the lazy thread migration is a bit better than the eager thread migration, but not significantly. Besides, a frequent migration increases the `Netperf` performance, since a monitor thread checks for outstanding interrupts more often.

In order to achieve these performances, it is necessary to send an interrupt notification as proposed in section 5.6. The benefit of this interrupt notification mechanism is outlined in Table 6.2. The throughput can be increased at least about 126 $\frac{Mbit}{s}$. A pol-

Migration frequency [Hz]	Netperf [$\frac{Mbit}{s}$] (eager)	Netperf [$\frac{Mbit}{s}$] (lazy)
0	846.25	846.25
1	841.40	844.67
10	841.77	844.91
20	846.51	849.92
100	855.64	858.84
1,000	855.72	859.12

Table 6.1: Eager vs. lazy guest user thread migration

icy migrating a vCPU up to a thousand times per second benefits the most. Thereby, lazy and eager migrations can profit to the same degree if one ignores the different performances between eager and lazy migrations.

f[a] [Hz]	Ne[b] [$\frac{Mbit}{s}$]	De[c] [$\frac{Mbit}{s}$]	Ine[d]	Nl[e] [$\frac{Mbit}{s}$]	Dl[f] [$\frac{Mbit}{s}$]	Inl[g]
1	714.50	126.90	127,412	711.14	133.53	129,535
10	712.74	129.03	126,957	709.65	135.26	128,003
20	715.04	131.47	125,893	715.13	134.79	131,813
100	727.98	127.66	113,892	724.31	134.53	119,083
1,000	686.79	168.93	71,879	674.15	184.97	77,869

[a] Migration frequency
[b] Netperf without interrupt notification (eager)
[c] Difference to Netperf with interrupt notification (eager)
[d] Interrupt notifications (eager)
[e] Netperf without interrupt notification (lazy)
[f] Difference to Netperf with interrupt notification (lazy)
[g] Interrupt notifications (lazy)

Table 6.2: Interrupt notification

To attain this performance, more than 70,000 cross processor IPCs are required. If a vCPU is migrated merely less than every 10 ms, more than 125,000 IPCs will be required. This is due to the fact that – currently or before the last reply – the vCPU handling the hardware interrupts is not executed by its initial monitor thread. Only if it is executed by its initial monitor thread a notification can be omitted. This applies more for frequent migrations.

We have considered the influence of the migration mechanism for the interrupt latency, in the next section we evaluate the overhead of the migration mechanism for building the Linux kernel.

6.2 Kernel Build Performance

Given that the network performance only indicates whether the interrupt latency has been increased significantly, the kernel build performance shows how far an application's execution time is affected by trying to swap a vCPU frequently. Since the performance difference between eager and lazy migrations is negligible, merely results of the lazy migration are outlined.

Migration frequency [Hz]	**Total execution time [s]**	**Performance loss [$\%$]**
0	187	0.0
1	191	2.1
10	191	2.1
20	193	3.2
100	199	6.4
1,000	209	11.8

Table 6.3: Kernel build performance

Table 6.3 shows that enforcing thousand migrations introduces a significant but acceptable overhead. Since the system boots from a RAM disk and has no hard disk, hardware interrupts can be excluded. Only the swapping mechanism is responsible for this performance degradation.

This performance degradation will probably be decreased if we apply our thermal balancing policy, since a migration is mostly performed if its gain is greater than the overhead introduced by the migration mechanism. Nevertheless, if our proposed load balancing policy does not reduce the migration frequency significantly, it will be necessary to migrate a vCPU less frequently. Thereby, one has to consider the tradeoff between the mispredicted power consumption of a core and the reduced overhead.

In the two previous sections we have evaluated the performance of our migration mechanism, in the next section we examine the impact of physical CPU dependent object accesses on the realization of a determined mapping.

6.3 Physical CPU Dependent Object Accesses

Our migration mechanism has been implemented on top of an IPC based system, therefore pCPU dependent objects are accessed frequently by a main thread and forbid a vCPU migration. To evaluate whether these pCPU dependent accesses prohibit to realize determined mappings, it has been counted how many cross processor IPCs need to be sent for realizing a mapping. Furthermore, it has been considered how often the mapping structure needs to be reset for realizing it. To get realistic values, the values have been accounted while building the Linux kernel.

As explained in subsection 4.6.3, to realize a mapping determined by a policy, it is often required to pass by the swapping method more than once. This can be seen in Table 6.4 for migration frequencies less than 100 Hz. Resetting the mapping structure only leads to the desired mapping in less than 40 % of these cases. Nevertheless, merely around 20 % of all cross processor IPCs indicate an outstanding swap, resulting in an unchanged mapping. Therefore, it is mandatory to reset the mapping structure as often as needed.

A migration frequency of 1,000 Hz exemplifies this. The mapping is only reset 4 % more often than a new mapping is set up. Consequently, only 66 % of the requested mappings have been realized. The breakeven point where each requested mapping can be realized is between a migration frequency of 20 and 100 Hz. Already 95 % of all requested migrations will be fulfilled, if a new mapping is set up each 10 ms. To achieve such a good coverage, the mapping structure needs to be reset more often than 125 % as a new mapping is determined. This is no major overhead for frequencies less

than 100 Hz, since the required cross processor IPCs are only a small fraction of IPCs that are required to indicate an outstanding interrupt.

	1 Hz	**10** Hz	**20** Hz	**100** Hz	**1,000** Hz
Et[a]	191	191	193	199	209
Dm[b]	191	1,913	3,867	19,994	209,648
Rm[c]	191	1,913	3,867	19,095	139,110
Sn[d]	244	2,361	4,929	23,610	159,452
Rtm[e]	550	4,800	10,163	45,538	217,848
Rr[f]	1.00	1.00	1.00	0.96	0.66
Mmr[g]	0.65	0.60	0.62	0.58	0.36
Rmr[h]	2.88	2.51	2.63	2.28	1.04
Snr[i]	0.22	0.19	0.22	0.19	0.13

[a] Execution time
[b] Different mappings
[c] Realized mappings
[d] Swap notification count
[e] Reseted mappings
[f] Realized-ratio: $\frac{c}{b}$
[g] Mappings-miss-ratio: $1 - \frac{c}{e}$
[h] Required-mappings-to-realized-ratio: $\frac{e}{c}$
[i] Swap-notification-miss-ratio: $1 - \frac{c}{d}$

Table 6.4: Physical CPU Dependent Object Accesses

We have shown that a vCPU can be migrated transparently without interfering a guest system with an acceptable performance degradation. Applying our proposed thermal balancing policy will probably decrease the performance degradation, since a vCPU migration will only be performed if the gain of a migration is greater than its overhead.

Bibliography

[1] Afterburner framework. http://l4ka.org/projects/virtualization/afterburn/.

[2] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 17–20 2000.

[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'03)*, New Orleans, LA, September 27 2003.

[4] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 260–270, New York, NY, USA, 2004. ACM Press.

[5] Kyeong-Jae Lee and Kevin Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11*, page 232.1, Washington, DC, USA, 2005. IEEE Computer Society.

[6] 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.

[7] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *First ACM SIGOPS EuroSys Conference*, Leuven, Belgium, April 18–21 2006.

[8] Netperf benchmark. http://www.netperf.org/.

[9] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *Operating Systems Review*, 41(3), July 2007.