Universität Karlsruhe
Fakultät für Informatik
Institut für Betriebs– und Dialogsysteme (IBDS)
Lehrstuhl Systemarchitektur

Balancing Power Consumption in Multiprocessor Systems

Diploma Thesis

Andreas Merkel

September 30, 2005

Advisors: Prof. Dr. Frank Bellosa
Dipl.–Inf. Andreas Weißel

Declaration
Erklärung

I hereby declare that I did this thesis on my own, using no other sources than the ones cited.

Hiermit erkläre ich, dass ich diese Arbeit selbstständig und ohne Verwendung anderer als der angegebenen Quellen angefertigt habe.

Andreas Merkel
Karlsruhe, September 30, 2005

Abstract

Actions usually taken to prevent processors from overheating, such as decreasing the frequency or stopping the execution flow, also degrade performance. Multiprocessor systems, however, offer the possibility of moving the task which caused a processor to overheat away to some other, cooler processor, so throttling becomes only a last resort taken if all of a system's processors are hot. Additionally, the different energy characteristics that different tasks are showing can be exploited, and hot tasks as well as cool tasks can be distributed evenly among all processors.

This work presents a mechanism for determining the energy characteristics of tasks by means of event monitoring counters, and an energy–aware scheduling policy, which strives to assign tasks to processors in a way that balances the power consumption of the processors, so individual processors do not overheat.

We implemented energy–aware scheduling for the Linux kernel. Evaluations show that the overhead incurred by the additional task migrations required for balancing power is negligible, and that, for many scenarios, energy–aware scheduling reduces the need for throttling and thus yields an increase in throughput.

Contents

1 Introduction

With increasing clock speed and circuit density, power dissipation has become an issue in todays high-performance microprocessors. Every Watt of electric power that a microprocessor consumes is transformed into heat and has to be removed from the system by a cooling facility. In the past, cooling facilities were designed for the worst case, i.e., a situation in which the processors are constantly operating at their theoretical maximum power. However, most ordinary tasks do not cause a processor to consume this maximum power. Therefore, designing cooling facilities for the worst case results in overprovisioning, and causes unnecessary costs.

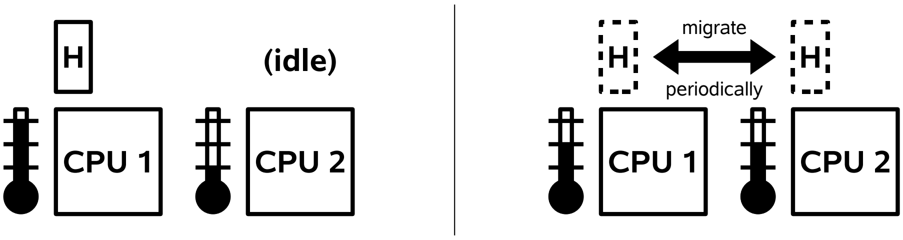
The alternative is to design cooling facilities for a more moderate thermal design power. If a processor executes tasks that exceed this power, it may reach a critical temperature, which can cause malfunctions or even physical damage. Hence, contemporary systems are monitoring the processor temperature and engage some throttling mechanism to keep the processor from overheating whenever a certain threshold temperature is reached.

The goal of throttling is to reduce the processor's power consumption. This can be achieved, for instance, by putting the processor in a low power sleep state for some time, by periodically disabling the clock signal (clock gating), by limiting the rate at which instructions are forwarded from the L1 cache or by reducing the processor's frequency and voltage. Besides reducing the power consumption of the processor, all of these measures also reduce the system's performance, because they delay the execution of instructions or reduce the frequency at which instructions are executed. Therefore, throttling should only be applied if really necessary.

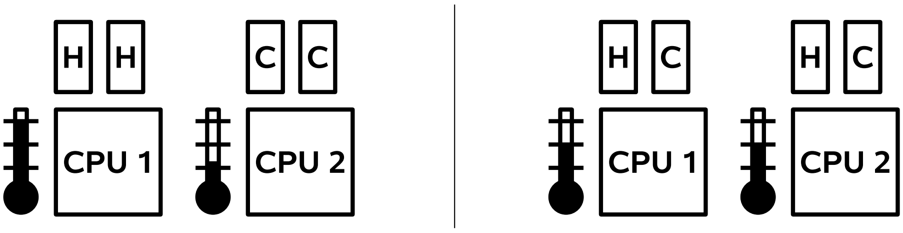
1.1 Thermal Imbalances

Idle processors consume considerably less power than busy processors. Modern processors offer special instructions to put the processor into a low power sleep state. The operating system executes these instructions whenever there is no work to be done for a CPU. This results in a lower power consumptions and a lower temperatures for idle processors.

Even two busy processors (of the same model) executing different tasks can show different power consumptions. The power consumption of recent processors depends on the type of instructions executed and therefore on the task that is running on a processor [BWWK03, IM03]. Different tasks utilize different kinds and different numbers of functional units on the processor chip. This results in individual tasks causing different power consumptions. *Hot tasks* cause a high power consumption and therefore



(a) Dual processor system running one hot task



(b) Dual processor system running two hot and two cool tasks

Figure 1.1: Examples of thermal imbalances

a high temperature of the processor; *cool tasks* consume less power, which leads to a lower processor temperature.

The following two examples elucidate how this can lead to *thermal imbalances*, i.e., the processors having different temperatures. In the first example, a dual processor system is running a single hot task (Figure 1.1(a)). If the task is running on the same CPU all the time (left side of the figure), the CPU concerned will have a high temperature, whereas the other CPU is idle and therefore cool. If the task were migrated to another CPU from time to time (right side of the figure), both CPUs could be kept to a medium temperature.

In the second example, two hot tasks and two cool tasks are running on a dual processor system (Figure 1.1(b)). If the two hot tasks are running together on one CPU, and the two cool tasks are running together on the other CPU (left side), this results in a high temperature for the first CPU and a low temperature for the second one. If each CPU were executing one hot and one cool task (right side), both CPUs would have medium temperatures.

In both examples, thermal imbalances, as shown on the left side of the figures, can lead to situations in which the *hot processors* operate at high temperatures and have to be throttled, whereas the temperature of the *cool processors* is far below the threshold temperature. However, as shown on the right side of the figures, thermal imbalances could be avoided if the right decisions concerning which task to run on what CPU were taken. Yet, schedulers found in contemporary operating systems are neither aware of the energy characteristics of the different tasks nor of the processors' temperatures, and thus do not take any precautions to prevent thermal imbalances.

1.2 Energy–Aware Scheduling

This thesis deals with operating system based dynamic thermal management in multiprocessor systems. Our goal is to reduce thermal imbalances between the processors of a system, so no individual processor becomes so hot it has to be throttled. Our approach takes advantage of cool processors and of the different energy characteristics of individual tasks. Instead of throttling hot processors, we move hot tasks that caused a processor to overheat away to a cool processor, and combine hot tasks with cool tasks on processors. By balancing the power consumption of the processors in the system, we keep all processors at the same temperature and avoid overheating individual processors. This reduces the need for throttling and therefore increases the system's performance.

Balancing the processors' power consumption requires assigning tasks to processors depending the tasks' energy characteristics and on the processors' temperatures. In an operating system, the scheduler is the component responsible for deciding which task runs on which processor. To get maximum performance out of a multiprocessor system under constraints imposed by the limited ability of each processor to dissipate energy without overheating, the scheduler has to know how much energy each task is consuming and how much energy can safely be dissipated on each processor per time unit, so it is able to make the right scheduling decisions and assign tasks to processors appropriately; it has to be *energy–aware*.

Schedulers found in contemporary operating systems try to maximize the throughput and the responsiveness perceived by the user. For making them energy–aware, they have to be extended to take the energy criterion into account, without neglecting their conventional criteria.

Therefore, we identify the following prerequisites: Firstly, we need a mechanism for determining the energy characteristics of a task, which means, how much energy a task is currently consuming and is expected to consume in the future, so we can identify hot tasks and cool tasks. Secondly, we need a policy for deciding which task shall run on which CPU, respecting the criteria of throughput, responsiveness, and energy.

Event monitoring counters included in modern microprocessors can be used to estimate the energy a processor spent during a certain period of time [BWWK03]. We use these counters to create *task energy profiles* describing the energy characteristics of the individual tasks. Our energy–aware scheduling policy is based on these profiles and strives to balance the processors' power consumption in order to minimize the need for throttling.

We implemented energy–aware scheduling for the Linux kernel and evaluated it under a number of different workloads. The evaluations show that energy–aware scheduling can significantly increase the throughput of a system.

1.3 Structure

The rest of this work is structured as follows: Chapter 2 discusses the fundamentals upon which our work depends and reviews related work. Chapter 3 analyzes the problem of thermal imbalances, which we already outlined in this chapter, more in depth, and presents the design of an energy–aware scheduling framework enabling the operating system to prevent thermal imbalances. Chapter 4 describes the changes done to Linux in order to integrate energy–aware scheduling and discusses some implementation issues. Chapter 5 shows evaluations of our Linux implementation of energy–aware scheduling using different workload scenarios. Chapter 6 concludes and discusses directions for future work.

2 Background and Related Work

2.1 Multiprocessor Scheduling

2.1.1 Affinity Scheduling

In SMP (simultaneous multiprocessing) systems, in principle, each task can run on each CPU. However, after a task has been running on a CPU for some time, it has warmed up the CPU's cache. Even if the scheduler suspends the execution of a task and assigns the CPU to another task, some of the first task's data is likely to be still in the cache when the scheduler switches back to it after some time. Therefore, it is beneficial to respect *processor affinity* and to resume a task on a CPU on which it ran previously, since this reduces the need for reloading the contents of the cache. *Affinity scheduling* [SL93, TTG95] takes advantage of this, and strives to avoid running tasks on different CPUs, either by giving tasks a priority bonus depending on the CPU, or by introducing local queues for the individual CPUs.

Our work is built upon the latter approach, which has been adopted by many of todays multiprocessor operating systems. Following this approach, the operating system divides the set of all runnable tasks in the system among the system's CPUs. It assigns a subset of the tasks to each CPU, and each CPU runs a scheduler of its own, which chooses the task to be run next on the CPU out of this subset. For each CPU, the scheduler organizes the subset of tasks that are eligible for execution on the CPU in some kind of data structure, e.g., a linked list, in order to keep track of the tasks. In Linux (and in other systems as well), this data structure is called the *runqueue*. Since every task belongs to one runqueue only, it is always executed on the same CPU unless it is transfered from one runqueue to another one.

The approach of local runqueues entails the problem of *load imbalances*, i.e., unequal runqueue lengths. The more tasks a runqueue consists of, the smaller is the share of CPU time each task gets. If there are CPUs having empty runqueues while the runqueues of other CPUs consist of multiple tasks, CPU capacity is wasted. Counteracting this problem requires *load balancing*, which means migrating tasks between runqueues.

General purpose operating systems strive to provide fairness to their tasks, and keep all runqueues to the same length by moving tasks from longer runqueue to shorter ones if necessary.

2.1.2 Hierarchical Load Balancing

NUMA (non uniform memory access) systems add a new dimension to the affinity problem. They introduce *node affinity*, which is given if a task is kept to the same node. Respecting node affinity is beneficial to performance [CDV$^+$94]. If a task gets migrated across the node boundary, the memory the task is referencing must either be transfered to the new node, or the task has to do inter–node accesses. Both possibilities incur performance penalties. Therefore, if load balancing can be done between CPUs belonging to the same node, this should be preferred to load balancing between CPUs belonging to different nodes.

Similarly, in SMT (simultaneous multithreading) systems, load balancing should preferably be done between sibling CPUs, since they share the same cache. In order to make the optimal load balancing decisions, the scheduler has to know about the CPU topology of the system, i.e., who is whose sibling and who shares the same node with whom. Linux introduces an abstraction called *scheduler domains* to represent this topology [Lin05].

A scheduler domain consists of two or more *CPU groups*. A CPU group is a set of CPUs. Scheduler domains are stacked in a hierarchical fashion to mirror the system's topology. The higher the level in this domain hierarchy, the costlier are the balancing operations in a domain.

An example (see Figure 2.1): In an eight–way SMP system consisting of two NUMA nodes with four processors each, and with each processor being two–way simultaneously multithreaded (16 logical CPUs in total), there are three levels in the domain hierarchy: On the lowest level, there are eight domains, each spanning one physical processor. Each of those domains consists of two CPU groups and each group spans one logical CPU. On the second level, there are two domains, each spanning one node. Both domains consist of four CPU groups. Each CPU group spans one physical processor and therefore consists of the processor's two logical CPUs. Finally, there is one top level domain spanning all CPUs. It consists of two CPU groups, each spanning one node. Each group consists of eight logical CPUs whose corresponding physical processors all reside on the same node.

With the domain hierarchy available, a scheduler can do *hierarchical load balancing* between the domain's groups. The goal of load balancing is to have the same runqueue length for each CPU. This is equivalent to having the same average runqueue length for all CPU groups on each level of the domain hierarchy. If group A has a greater average runqueue length than group B, tasks have to be migrated from some CPU in group A to some CPU in group B. Since the cost of migrations rises with the level in the domain hierarchy, a hierarchical load balancer solves imbalances on the lowest level possible.

For the example topology above, hierarchical load balancing means resolving imbalances between two sibling logical CPUs by transferring tasks from one sibling to the other, instead of moving tasks from a different physical CPU to the first sibling or moving tasks to a different physical CPU from the second sibling. Similarly, tasks are

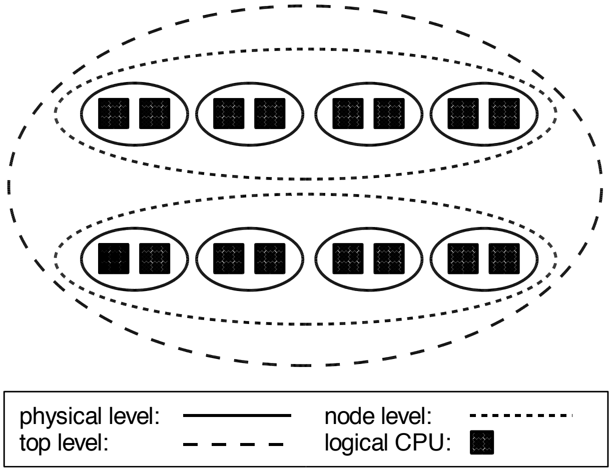


Figure 2.1: Example of scheduler domains

only transfered between CPUs residing on different nodes if the top–level domain is unbalanced.

2.2 Energy and Temperature Estimation

2.2.1 Online Energy Estimation Using Event Monitoring Counters

Modern processors like the Pentium 4 offer special registers called *performance monitoring counters*, or more general, *event monitoring counters*. These counters are intended for delivering values useful for performance analysis and optimization, and therefore are able to count various processor–internal events. As those events correspond to activities on the processor chip, the operating system can use these counters for estimating the energy the processor spends during a certain period of time [BWWK03, Kel03].

Assuming that the processor consumes a certain fixed amount of energy for each activity, the energy consumption of the processor can be estimated as follows: A set of n events that can be counted simultaneously is chosen, and a fixed amount of energy a_i is associated with each event. The energy spent during a period of time is calculated as the linear combination of the counter values c_i:

$$E = \sum_{i=1}^{n} a_i \cdot c_i \tag{2.1}$$

The weights a_i are calibrated by measuring the real energy consumption with a multimeter for several test applications, counting the events that occur during the test runs, and solving the resulting linear equations.

Assigning a constant amount of energy to each event neglects the fact that the power consumption of a processor depends on temperature. Especially leakage power increases with higher temperatures. However, within the temperature window of 20 degrees, in which we operate (see Chapter 5), the influence of temperature on energy consumption is only small. This justifies neglecting temperature effects.

Since event monitoring counters are implemented as registers in hardware, they can be read with low overhead. This allows to estimate the energy consumption very frequently and to determine the energy consumed during a period as short as a scheduling interval. Therefore, it is possible to attribute energy consumption to distinct tasks, which is a prerequisite for determining the individual tasks' energy characteristics.

The mechanism for online energy estimation just described has been implemented for the Intel Pentium 4 Processor and the Linux kernel [Kel03] and yields an estimation error of less than 6% for real–world integer applications. Since the counters present in the Pentium 4 do not completely cover all relevant events corresponding to floating point activities, the error is higher for floating point applications (up to 20%).

This work uses the mechanism for energy estimation described, but extends it to estimate energy consumption separately for the logical CPUs of a simultaneously multithreaded processor (see Section 3.4.3 in the next chapter).

2.2.2 Thermal Model of a Microprocessor

When the energy consumption of the processor is known, it can be used as an input for a thermal model of the processor and its cooling facility [BWWK03, Kel03]. Using such a model, the temperature of the processor can be estimated quite accurately. The error resulting from estimating energy and then estimating temperature from the energy estimate is smaller than one degree for real–world applications. Conversely, the thermal model can be used to calculate the amount of energy which may be dissipated during a certain period of time without overheating the processor. This information is of vital interest for an energy–aware scheduler.

The energy that a processor consumes gets dissipated in form of heat. This energy is stored in the chip and the heat sink and results in increased temperature of those components. But the energy is not stored forever: Owing to the thermal conductivity of the heat sink, the energy is delivered to the surrounding air and removed by the air flow that the fans generate.

If the processor operates at a static power, dissipation and conduction on the long run form an equilibrium, which results in a constant temperature. On the other hand, if there is a change in power, this results in a change in temperature. Because of the thermal capacitance of the heat sink, this change does not take place immediately. On an increase in power, the temperature rises exponentially until the equilibrium resettles at a higher temperature. Similarly, on a decrease in power, the temperature falls expo-

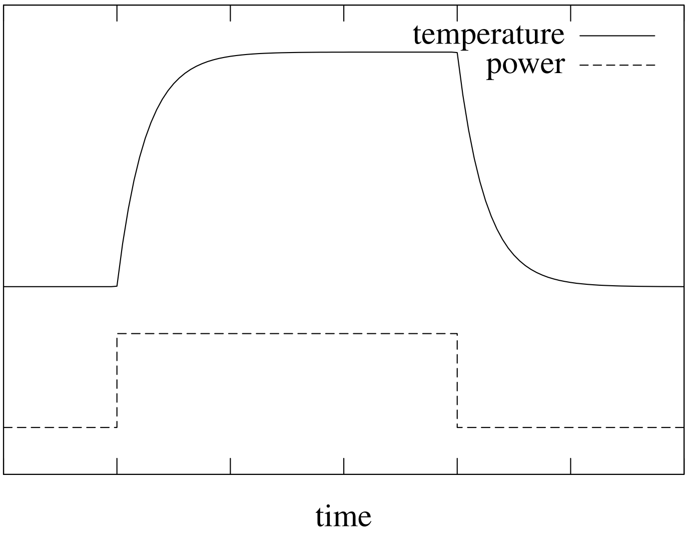


Figure 2.2: Dependence of chip temperature from power consumption

nentially. Figure 2.2 illustrates this: After one time unit, the formerly constant power raises to a higher level for four time units. Then it drops back to its original level. The temperature follows power, but does not change at once, but exponentially over time.

The course of the processor temperature over time is modeled with the following equation [BWWK03]:

$$\vartheta(t) = \frac{-\tilde{c}}{c_2} \cdot e^{-c_2 t} + \frac{c_1}{c_2} \cdot P + \vartheta_0 \tag{2.2}$$

ϑ_0 is the ambient temperature. c_1 and c_2 are constants depending on the thermal resistance and the thermal capacitance of the processor and the heat sink. $\tilde{c}$ is an integration constant depending on the temperature of the processor at the time $t = 0$.

Equation 2.2 consists of a dynamic part (first addend) and a static part (second and third addend). The static part depends only on the ambient temperature and the power consumption of the processor. It models the temperature that the processor reaches after operating with constant power P for a long time. The dynamic part is an exponential function with time constant c_2 and models the exponential rise or decay of the processor temperature after a change in power consumption.

The time constant of this exponential function can be determined by starting a task which produces a maximum of heat on a processor formerly idle, recording the temperature values over time, and fitting an exponential function to the experimental data. With knowledge of c_2, the constant c_1 can then be determined from the static part of Equation 2.2 by simultaneously measuring temperature and power consumption while the CPU is executing workloads with different but constant power consumptions.

The ability of different materials to store and to conduct heat can be seen analogously to the corresponding electrical qualities, electrical capacitance and electrical conductivity (or its inverse, electrical resistance). From this point of view, the thermal model of the processor and the heat sink just described corresponds to a *compact model* [HGS$^+$04] consisting of one thermal capacitor and one thermal resistor. More elaborate compact models consisting of greater numbers of thermal capacitors and resistors, which characterize the thermal properties of a processor chip more accurately and yield different temperatures for the individual functional units of a chip, have been proposed [LS05]. However, considering more than one temperature value per chip for scheduling is beyond the scope of this work.

2.3 Energy–Aware Scheduling

An energy–aware scheduler considers the energy characteristics of individual tasks and the temperature of the processors when making scheduling decisions. Up to now, most of the work on energy–aware scheduling does not take multiple processors into account, but is focused on uniprocessors. There are also some hardware related approaches, which, like our approach, mitigate thermal imbalances by migrating computation.

Process Cruise Control

On processors supporting frequency and voltage scaling, the scheduler can use the energy characteristics of a task to determine the optimal frequency at which the processor should run the task [WB02]. The scheduler uses a task's IPC (instructions per cycle) value and the frequency of memory accesses to determine the optimal frequency at which energy consumption is reduced without decreasing performance substantially. However, frequency and voltage scaling are not available on most of todays high performance processors used in multiprocessor server machines.

Task Level Throttling

If the operating system knows which tasks are responsible for a rise in processor temperature, it is able to keep the temperature below a certain limit by throttling only the hot tasks, instead of penalizing all tasks [RS99]. The scheduler identifies hot tasks by their high CPU activity and throttles them by executing the `hlt` instruction, which puts the processor in a low power state until the next interrupt arrives.

We argue that in multiprocessor systems, if there are cooler processors, migrating a hot task to such a processor is superior to throttling. Additionally, considering only CPU activity has proven insufficient for determining a task's energy characteristic on recent processors like the Pentium 4 [BWWK03].

Energy Containers

The abstraction of resource containers [BDM99] allows the management of energy as a first class resource in distributed systems [Wai03, WB04]. Each task has an associated energy container on which the operating system accounts the energy consumed by the task. Only tasks whose energy container is non–empty may be chosen by the scheduler. Since this work's focus is not on choosing which tasks are allowed to be scheduled, but rather on where they are scheduled, the two approaches are fully compatible and could easily be combined.

Heat–And–Run

The principle of moving computation away when temperature gets too high is part of the *Heat–and–Run* scheduling policy developed for simultaneously multithreaded chip multiprocessors [GPV04]. Heat–and–Run characterizes tasks by their IPC value and uses this characterization to co–schedule tasks on sibling processors in a way that produces maximum heat. If a certain temperature limit is reached, the tasks are migrated to another core on the chip multiprocessor. However, Heat–and–Run was not implemented on real hardware, but uses the *Wattch* [BTM00] architectural–level simulator.

Activity Migration

A more hardware related approach to dynamic thermal management consists in including spare resources like register files, ALUs, or issue queues on the processor chip and migrating computation to one of those spare resources when the original resource reaches a critical temperature [HBA03,SSH$^+$03]. Our approach works on a more coarse grained level, moving computation not within a chip but between chips. Also, in our approach, decisions concerning the migration of tasks are taken at the operating system level, whereas activity migration within a chip can only be triggered by a hardware mechanism, since the microarchitecture of the processor is transparent to software. The hardware, however, has no knowledge about tasks and can only migrate computation in general, but not on a per task basis.

Energy–Aware Fetch Policy

Another approach, which also operates at the hardware level, modifies the fetch policy of a SMT chip [DM05]. Instead of characterizing individual tasks, this approach characterizes the individual hardware threads of a SMT processor by their utilization of the chip's functional units. Depending on the current temperatures of the functional units, the processor prefers fetching instructions for threads that are mainly using units that are currently cool.

3 Energy–Aware Scheduling

3.1 Problem Analysis

3.1.1 Thermal Imbalances

The power consumption of a modern microprocessor depends strongly on the type of instructions it executes and thus on the program code of the task that runs on the processor [BWWK03, IM03]. On the $2.2GHz$ Pentium 4 Xeon processor we used in our experiments (see Chapter 5), we measured power consumptions ranging from $38W$ to $61W$ for different compute intensive tasks.

As already stated in the introduction, schedulers found in contemporary operating systems do not take the tasks' different energy characteristics into account when deciding on which CPU of a multiprocessor system to run a task. As a result, most of the time, the processors have different power consumptions. Since the energy that a processor consumes determines the temperature of the processor chip (see Section 2.2.2), this results in different temperatures for the different processors of a system. This is even more the case if some CPUs of a system are idle, since idle processors consume considerably less power than busy processors.

Besides power consumption, the temperature of a processor also depends on the cooling facilities: Processor A might be located closer to some cooling facility like a fan or an air intake than processor B. Therefore, even if processors A and B are operating at the same power, A's temperature will be lower than B's. This also means that processor A is able sustain a higher power consumption without overheating than processor B.

So we can identify two reasons for thermal imbalances: power consumption and cooling properties. Power consumption is highly dynamic. It depends on the tasks that run on a CPU and can therefore — at least to some degree — be influenced by the operating system via scheduling decisions. Opposed to this, we treat the cooling properties of a processor as static. Although modern cooling systems allow properties like the fan speed to be controlled by software, the maximum power each processor can dissipate without overheating, which is the critical property in our case, is fixed.

3.1.2 Timescales

As we depicted in Section 2.2.2, when the power consumption of a processor changes, the chip temperature does not change immediately, but continuously over time. Although there are certain variations in the power consumption when a processor executes a particular task, the greatest changes in power consumption occur when the operating system switches the CPU between tasks with different energy characteristics, when a busy CPU becomes idle, and when an idle CPU becomes busy.

When the power consumption of a processor changes, it takes seconds up to minutes for the temperature to change significantly because of the high thermal capacitance of the processor chip and the heat sink. Compared to this, scheduling intervals are short — the length of a timeslice ranges between 10 milliseconds and 100 milliseconds in general–purpose operating systems. During such a short period of time, the processor hardly heats up or cools down at all. Therefore, if a processor runs several tasks with different energy characteristics that are scheduled in round–robin fashion, the resulting temperature is a mix of the individual tasks' characteristics. Hence we can avoid overheating a processor running hot tasks, if we run some cool tasks on the same processor, too.

On the other hand, since processors only heat up slowly, even if a processor is running only hot tasks, it takes some time until the processor reaches a critical temperature. If a hot task is started on or migrated to a cool processor, it can run there for some seconds or even minutes before overheating the processor. Therefore, we can also avoid overheating processors by migrating hot tasks to cool processors from time to time.

3.1.3 Need for On–Line Energy Estimation

The knowledge about the tasks' energy characteristics is a prerequisite for influencing the processors' temperatures via scheduling decisions. Therefore, the operating system must be able to identify hot tasks and cool tasks. An analysis of the processor's power consumption while running a particular task shows that power consumption is fairly static most of the time, but exhibits changes as the task experiences different phases of execution, e.g. runs different algorithms successively [Bel01]. The sequence and the duration of these phases depend on the task's input data. Therefore, the energy characteristics of a task cannot be known in advance and thus cannot be determined by off–line analysis.

Characterizing tasks on–line means we must observe a certain quality that represents the tasks' characteristics at least once every timeslice, since the CPU might be executing a different task every timeslice. This is the reason why we cannot characterize individual tasks by the temperature of the processor: As described in the preceding subsection, a processor's temperature only changes slowly. It is possible that a processor currently executing a cool task is hot, since it has executed hot tasks in the past. Similarly, as already mentioned, a processor executing hot tasks in turn with cool tasks has a medium temperature.

Unlike temperature, the energy consumed by the processor is suitable for characterizing individual tasks on–line. Individual tasks cause typical, distinguishable power consumptions of the processor during the timeslices of their execution. Therefore, we can characterize individual tasks if we can determine the amount of energy the processor consumes during each timeslice. Although there is measurement hardware with a temporal resolution high enough to determine these amounts of energy [FS99], such hardware is too expensive for use in production environments. Also, the sampling and the evaluation of on–line measurements incurs high overhead, and is therefore usually done by a second computer system.

In contrast to this, on–line energy estimation by means of event monitoring counters as described in Section 2.2.1 is a convenient way for the operating system to determine the amount of energy the processor consumes during each timeslice. Without requiring additional hardware, this mechanism delivers quite accurate results. Most important, the overhead for reading the event counters and calculating the energy estimation is low, so the operating system can estimate the energy consumption every timeslice.

3.1.4 SMT Issues

Recent processors feature *simultaneous multithreading* (SMT), the ability to run several (logical) threads of control on a single (physical) processor. This poses additional challenges to energy–aware scheduling.

To the software, and thus to the operating system, a SMT processor looks like several processors (also called *logical processors*) executing different tasks independently. On the hardware side, however, there is only one chip. The logical processors all share the same physical resources on the chip and therefore all dissipate their heat at the same locations. Hence, if tasks with different energy characteristics are running on different logical processors in parallel, the resulting chip temperature is a mix of the individual tasks' characteristics, similar to the situation described above, with several tasks running on the same processor in round robin fashion.

Since the threads of a SMT processor run truly in parallel, not only the chip's temperature, but also its power consumption is a mix of the characteristics of the tasks that are running in parallel on the logical processors. To characterize individual tasks, the operating system needs a way to determine which share of the processor's power consumption is caused by which of its logical threads. Energy estimation by means of event monitoring counters can accomplish this, if the processor's event monitoring counters are able to distinguish which of the processor's threads caused an event. In this case, energy estimation can take place separately for each logical processor, and power consumption can be attributed to individual tasks running in parallel on a SMT processor.

3.2 Approach

To tackle the problem of thermal imbalances and the resulting need for throttling some processors of a system, we introduce an energy–aware scheduling framework. We enable the operating system to know about the energy characteristics of the tasks it manages, and to choose a suitable CPU for running each task respecting the processors' temperatures and abilities to dissipate heat.

However, our approach does not directly deal with throttling. We assume that some other authority decides when a processor is throttled and when not, depending on the processor's temperature. Since temperature is the trigger for throttling, our energy–aware scheduler strives to assign tasks to CPUs in a way that keeps all CPUs' temperatures below the limit temperature at which throttling is engaged. This way, we can minimize the need for throttling.

Since it is not always possible to keep temperature below the maximum temperature by means of scheduling, e.g if there are only hot tasks, energy–aware scheduling is only a best effort approach. Limiting temperature remains the responsibility of the throttling component, whereas the energy–aware scheduler strives to leave as few work as possible to this component.

Our approach does not depend on a specific operating system or architecture. The only prerequisite from the operating system side is support for CPU affinity scheduling and load balancing as described in Section 2.1.1. To work on SMT processors, our approach additionally requires some sort of hierarchical load balancing that is able to distinguish between physical and logical CPUs, like scheduler domains introduced in Section 2.1.2. From the hardware side, the only prerequisite is support for event monitoring counters. On SMT processors, at least some of the counters must be able to attribute events to distinct logical CPUs .

3.3 Overall Design

We propose an energy–aware scheduling framework consisting of three components — energy estimator, energy profiler and energy–aware scheduler — to incorporate energy–aware scheduling into an operating system. Figure 3.1 depicts how these components interact. The energy–aware scheduler is the heart of the framework. It pursues an energy–aware scheduling policy whose goal is to avoid overheating processors. To achieve this goal, the scheduler assigns tasks to CPUs in an appropriate way. This requires knowledge about the energy characteristics of the individual tasks and about the power consumption and temperature of each processor.

The *energy profiler* is the component responsible for characterizing tasks by their power consumption. For each task, it maintains an *energy profile* which describes the task's energy characteristics. Besides the task–specific energy profiles, the energy profiler also calculates CPU–specific power consumption rates. Both the task energy profiles and the CPU power consumption rates are the base for the energy–aware scheduler's decisions.

The energy profiler needs knowledge about the amount of energy each CPU consumes while executing a task for one timeslice to calculate the task energy profiles and the CPU power consumption rates. These energy values are provided by the third

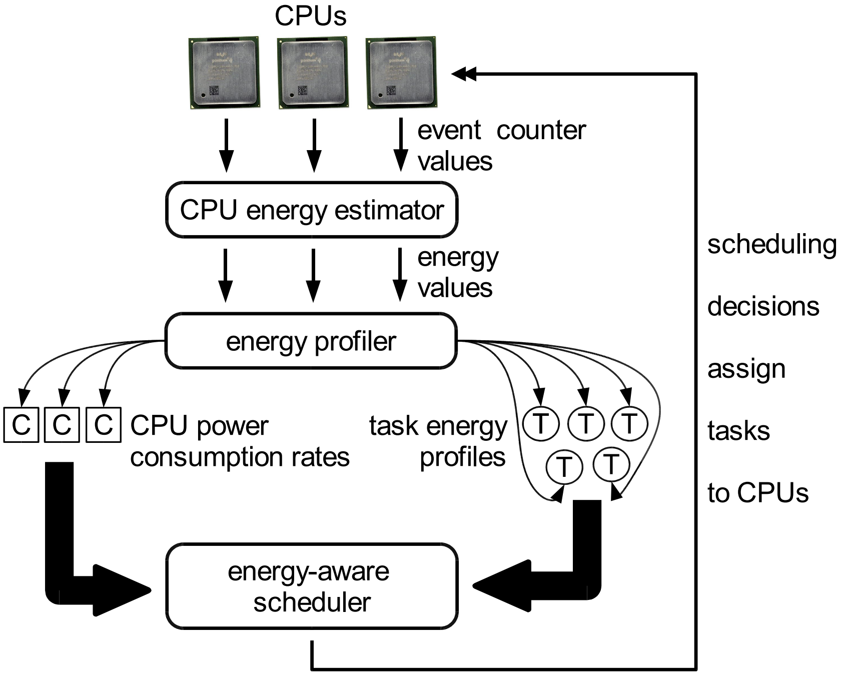


Figure 3.1: Overall design

component of our framework, the *energy estimator*. Using the CPUs' event monitoring counters, the energy estimator assesses the energy consumed by each CPU.

The remainder of this chapter discusses these three components from a bottom–up view, beginning with the energy estimator and finishing with the energy–aware scheduler. As mentioned in the preceding section, our design is not operating system or platform specific. At some places in the following sections, we use Linux or the Pentium 4 architecture as an example. An actual implementation of energy–aware scheduling for Linux and the Pentium 4 platform is described in the next chapter.

3.4 Energy Estimator

3.4.1 Motivation

The energy profiler needs to determine how much energy each of the tasks running on a CPU is consuming (see Section 3.5). Since a CPU executes the tasks in its runqueue in round robin fashion, each one for a certain period of time, the energy profiler needs knowledge about the amount of energy the CPU consumes during these periods of time. The energy estimator is the component which provides this information. As already mentioned in Section 3.1.3, we choose to determine the processors' energy consumption using event monitoring counters, because access to those counters is fast and allows to attribute the energy a processor consumes exactly to the different timeslices during which different tasks are executed.

3.4.2 Principle of Operation

The energy estimator transforms event counter values read from a CPU to energy values. It uses the estimation mechanism described in Section 2.2.1.

The energy estimator offers an interface to the energy profiler. Whenever the profiler invokes the estimator using this interface, the estimator reads the CPU's event monitoring counters and compares them to the values the counters had at the last invocation. It calculates the difference for each event, and weights it with the amount of energy corresponding to that type of event. The result, the energy consumed by the CPU during the time between the two invocations, is returned to the energy profiler.

3.4.3 SMT Issues

For characterizing tasks running on a SMT processor, we need to attribute the events that occur on the processor to the tasks running on its logical CPUs. However, it may not be possible to count all events individually for each logical CPU. For certain events, the counting hardware is unable to distinguish which logical CPU has caused the event. Furthermore, on the two–way multithreaded Pentium 4 architecture, for which we implemented the energy estimator, there is only one set of event monitoring registers shared by both logical CPUs. Therefore, even some of the events that could be attributed to distinct logical CPUs, cannot be counted simultaneously for both CPUs due to a lack of counter registers [Int03].

For events that cannot be counted on a per–logical–CPU basis, there is no other way than to divide the number of events by two and to assign half of the events to each logical CPU, which may cause errors in the energy estimation. Events that cannot be counted simultaneously for both logical CPUs due to a lack of counter registers could be handled the same way by instructing one counter register to count events for both processors and by dividing the value of the register between the processors. We choose a different approach for handling these events, which lessens the error in energy estimation: The energy estimator multiplexes counter registers between both logical CPUs. Every timer tick, it instructs the counter to count events for a different logical CPU. This way, the counter counts events for a logical CPU every other timer tick. The energy estimator doubles the values obtained this way to compensate for counting events only half of the time for each CPU.

So in summary, we can divide the events into three classes:

- Events that can be counted for individual logical CPUs, and for which there is a separate counter register for each logical CPU. This is the ideal case.
- Events that can be counted for individual logical CPUs, but for which there is only one counter register. In this case, we multiplex the register.

- Events that cannot be counted for individual logical CPUs. This is the worst case, since we have to divide the counter value between the logical CPUs.

3.5 Energy Profiler

3.5.1 Motivation

An energy–aware scheduler needs to know about the energy characteristics of the individual tasks, i.e. how much energy a task is currently consuming per time unit, and how much energy it is expected to be consuming in the near future in order to be able to make the right decisions concerning which task to run on which CPU. The energy profiler, which we introduce in the following sections, is the component responsible for doing an online analysis of the tasks' energy consumption.

Besides knowing how much energy is consumed on a per–task basis, the scheduler also needs to know how much energy each processor is consuming over time. Since a processor's temperature is determined by the amount of energy the processor consumed in the past, past energy consumption is a valuable information for the scheduler, too. For each CPU, the energy profiler therefore maintains two power consumption rates representing present and past energy consumption on a per–CPU basis.

3.5.2 Task Energy Profiles

For characterizing individual tasks by their energy consumption, we introduce *task energy profiles*. A task's energy profile is part of the task's runtime context and describes the task's current energy characteristics. It consists of an energy value which is an estimation for the energy the task will consume if it is allowed to run on a CPU for one timeslice. The energy profiler maintains an energy profile for each task. The scheduler uses the information from the profiles for deciding on which CPU to run which task (see Section 3.6).

To make the optimal decision regarding on which CPU to run a task for the next time, the scheduler would have to know how much energy the task is going to consume during its next timeslice. This is not possible, since in systems where input data influence the behavior of tasks in a non–deterministic way, the energy profiler cannot look into the future. Although the energy characteristics of a task depend on the actual piece of program code being executed and thus may change over time, most tasks exhibit a fairly static behavior. This makes the amount of energy a task consumed the last time it was executed a good guess for the amount of energy the task will consume the next time it is allowed to run.

Considering the consequences a change in a task's energy profile might have, namely the transfer of the task to another CPU (see Section 3.6), we should avoid changing a task's energy profile too often and too drastically. Therefore, we do not only take the energy spent during the last timeslice into account, but use an exponential average.

In the area of power management, exponential averages have been used to smooth data gathered about the performance demands of applications [GCW95,PBB98,FM02]. We use it to smooth out the energy profiles over time. The exponential average is a weighted moving average intended for calculating the average of a value which is sampled over constant periods of time. It weights recent sample values with a higher weight than sample values lying further in the past. Since on every iteration, the past values are weighted down anew, their contribution to the exponential average fades over time. As a result, short term changes in a task's behavior do not cause the task's energy profile to change significantly, whereas a permanent change is reflected in the energy profile after an appropriate time.

Whenever a task's timeslice is over, the energy profiler queries the energy estimator about the amount of energy the CPU consumed during the timeslice and updates the task's energy profile by recalculating the exponential average. Since tasks may also block, release the CPU voluntarily , or, in preemptively scheduled systems, be deprived of it by a higher priority task in the middle of a timeslice, the energy profiler has to do the same querying and calculation on such occasions. However, it has to be aware that the resulting energy value is not the energy spent during a full timeslice, but during a shorter period of time.

3.5.3 Exponential Average for Variable Sampling Periods

Exponential average is intended for calculating the average of a value sampled over constant periods of time. In our case, this period usually corresponds to one timeslice. But as mentioned in the last subsection, sometimes a task does not execute for a full timeslice. Additionally, some operating systems, like Linux, use different timeslice lengths for different tasks, depending on a task's priority. Therefore, the length of the sampling period is no longer a constant.

There are two solutions to this problem: Firstly, we can shorten the sampling period, so it is considerably smaller than the mean time between two subsequent task switches. This way, we recalculate the exponential average not only at the end of each timeslice, but, for instance, on every timer tick, so the energy profile of a task is up to date even if the task stops executing in the middle of a timeslice. Secondly, we can extend the exponential averaging algorithm to support variable periods, so we are able to calculate the exponential average at any time a task stops executing. We choose the latter approach, since it incurs less overhead (we have to recalculate the exponential average and to query the energy estimator less often) and is more exact (a task switch might occur even between two timer ticks).

We start from the conventional exponential average: The exponential average x_n for sampling period n is calculated by weighting the current sample value v_n, which in our case is the energy spent during the sampling period, with a weight p between 0 and 1, while the exponential average x_{n-1} of the last sampling period is weighted with $(1-p)$:

$$x_n = p \cdot v_n + (1-p) \cdot x_{n-1} \tag{3.1}$$

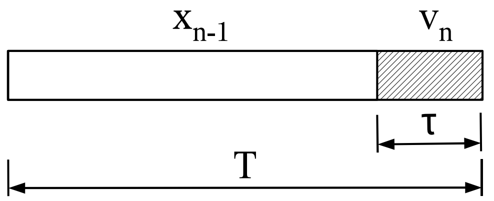


Figure 3.2: Calculating an exponential average

We can interpret the weight p the following way (also see Figure 3.2): v_n is the energy spent during a sampling period of constant duration τ. We calculate the average over a larger super period with duration T, so we can calculate the weight p as the quotient $(\tau\ /\ T)$. We take the old exponential average x_{n-1} as the value for the energy spent during the rest of the super period $(T-\tau)$ and weight it with $(1-p) = ((T-\tau)\ /\ T)$. This yields

$$x_n = \frac{\tau}{T} \cdot v_n + \frac{T-\tau}{T} \cdot x_{n-1} \tag{3.2}$$

If the lengths of the sampling periods differ, we can now calculate the exponential average after a sampling period with length τ_n by replacing τ with τ_n:

$$x_n = \frac{\tau_n}{T} \cdot v_n + \frac{T-\tau_n}{T} \cdot x_{n-1} \tag{3.3}$$

If τ_n is shorter than τ, we give the past a bigger weight, which compensates for calculating the exponential average more often. (The past values are weighted down with every iteration.) Vice versa, if τ_n is longer, we give the past a smaller weight, but calculate the average less often.

However, Equation 3.3 is applicable only if the sample value v_n is independent of the length of the sampling period τ_n, as is the case, e.g., with temperature values. But the energy spent is not independent from the sampling period: The longer the period, the more energy is spent. Therefore, we have to scale up or scale down the energy value v_n if τ_n is shorter respectively longer than τ. We accomplish this by multiplying v_n with $(\tau\ /\ \tau_n)$, so the final equation used for calculating the energy profiles is

$$x_n = \frac{\tau}{T} \cdot v_n + \frac{T-\tau_n}{T} \cdot x_{n-1} \tag{3.4}$$

where τ is the length of a standard timeslice and τ_n is the actual execution time. The value of T must be chosen depending on how quickly the exponential average is supposed to react on changes in the input values v_n. With a standard timeslice length of $100ms$, $1.6s$ (16 timeslices) turned out to be a good choice for T.

3.5.4 Newly Started Tasks

Since we want to have energy profiles quickly available for newly started tasks, we use smaller values for tasks which have just been started. If a task has not yet been executing for $1.6s$, we use the execution time of the task as the value for T. This way, the energy profile adapts quickly to the energy characteristics of newly started tasks, which is especially important for tasks that run only for a short time.

We can avoid unnecessary task migrations if the scheduler knows about the energy characteristics of a task even before the task is started. This way, the scheduler can place the task on the right CPU from the beginning. For workloads consisting of many short running tasks, this is even a prerequisite for energy–aware scheduling to work: A short running task might already have terminated before an energy–profile is available via online analysis.

As stated in Section 3.1.3, we cannot know about the energy characteristics of a task in advance because of the different phases occurring during the task run which depend on input data. Most tasks, however, do some initialization before processing any input data. Therefore, if a task is started, its initial behavior is independent of the data that the task processes.

The energy profiler remembers the amount of energy a newly started task consumes during its first timeslice to take advantage of this. If a task is started again from the same binary, the energy profiler uses the remembered value as starting value for the energy profile. For binaries started for the very first time, it uses a default value.

3.5.5 Calculated Power Consumption Rate

Next to the task–specific energy profiles, the energy profiler also maintains CPU–specific power consumption rates. First of all, the scheduler has to know how much power is currently consumed by each CPU, because (as we will motivate in the following section), power consumption should be balanced between the CPUs. In principle, on a system which schedules the tasks a runqueue consists of in round robin fashion, the power consumption of a CPU can be calculated by averaging the values taken from the energy profiles of all tasks in the CPU's runqueue. Only this average matters, since, as mentioned in Section 3.1.2, scheduling intervals are much shorter than the time it takes for the processor's temperature to change.

However, the *calculated power consumption rate* obtained this way is only of limited accuracy. The timeslice length might be different for different tasks; higher priority tasks might get longer timeslices, as it is the case in Linux. Additionally, the energy profiler cannot know whether each task in the runqueue will fully utilize its timeslice, or if some tasks will block in the middle of a timeslice and no longer be runnable.

Another shortcoming of the calculated consumption rate is that it only considers tasks that are currently members of a CPU's runqueue. This is insufficient for two reasons: Firstly, there may be tasks which are blocking and unblocking frequently, e.g. a web server answering requests and sleeping during the time between successive requests. A blocked task is waiting for some event and thus not eligible by the scheduler and no member of the runqueue. Since such tasks usually resume executing on the same CPU, they should also be considered when they are blocked and currently not members of the CPU's runqueue. Secondly, the energy dissipated by tasks which were executing on the CPU in the past is still partly stored in the processor chip and the heat sink. The scheduler must therefore not only consider the processor's current power consumption, but also its temperature, which is determined by past power consumption. Otherwise, there would be no way for the scheduler to distinguish between a cool processor and a hot one whose hot tasks have just been migrated away (other than measuring temperature, which is costly).

3.5.6 Empirical Power Consumption Rate

To overcome the shortcomings of the calculated power consumption rate, we introduce a second CPU–specific power consumption rate, which is also maintained by the energy profiler. The profiler determines this rate in an empirical way by looking at the amount of energy a CPU really consumes during each timeslice; therefore we call it *empirical power consumption rate*. This empirical consumption rate mirrors the current processor temperature. Hence, it must consider the energy consumed in the past to a degree that corresponds to that part of the energy that is still stored in the processor and the heat sink.

As Section 2.2.2 elucidates, the energy stored in the processor and the heat sink manifests itself in form of heat and raises respectively decays exponentially. There is a close relationship between an exponential function and an exponential average. Since, when calculating an exponential average, past sample values are multiplied with $(1-p)$ every iteration (compare Equation 3.1), their contribution fades exponentially: A value that was sampled m iterations ago is weighted with $(1-p)^m$.

Therefore, we use an exponential average to calculate the empirical power consumption rate. Whenever the energy profiler queries the energy estimator to update the energy profile of the task that was executing during the last timeslice, it uses the energy value delivered by the energy estimator also to calculate a CPU–specific exponential average. This exponential average is similar to the task energy profiles, but considers any task that is running on the CPU in question instead of being task specific.

Since the empirical consumption rate is determined empirically by looking at the amount of energy a task consumed, it only considers tasks to the extent they really run on the CPU, i.e., not necessarily to the extent of a full timeslice. Since we calculate an exponential average, the empirical consumption rate also considers tasks which ran in the past but are no longer members of the runqueue.

The empirical consumption rate mirrors the energy that was dissipated by tasks in the past is still stored in the processor and the heat sink and therefore models the processor's temperature. Hence, we fit the exponential average used to calculate this consumption rate to the exponential function of our thermal model (Equation 2.2). If a hot task has just been migrated away from a processor and the temperature of the processor as well as the empirical consumption rate are at their upper limits, the consumption rate should be halfway down to the power consumption of an idle processor when the processor's temperature is halfway down to the temperature of a processor that has been idle for a long time. We accomplish this by choosing a super period length for the exponential average that corresponds to the time constant c_2 in Equation 2.2. Appendix A.1 elucidates how we derive the super period length from the time constant. For brevity, we only quote the result here:

$$T = \frac{\tau}{1 - e^{c_2 \tau}} \tag{3.5}$$

However, the empirical power consumption rate has one disadvantage: A task that has been newly migrated to a CPU can be considered immediately by the calculated consumption rate, whereas the empirical rate does not consider it until it has used up its timeslice for the first time. This is important when the scheduler balances power consumption by moving tasks from one CPU to another, since it avoids over–balancing, i.e. moving over too many tasks so one imbalance is replaced by another imbalance into the opposite direction. The energy profiler therefore maintains both, calculated and empirical power consumption rate, for each CPU, so the scheduler can consider both rates for its decisions.

3.5.7 SMT Issues

The energy a task consumes depends heavily on whether it is executed alone on a physical processor or together with other tasks running on sibling logical CPUs. A task running alone executes considerably more instructions per time unit and is therefore consuming more power. Also, different combinations of tasks in many cases yield different power consumptions for each of the tasks, depending on whether they use mainly the same CPU resources and thus obstruct each other, or different ones.

The energy profiler cannot compensate this, since it calculates the task energy profiles based on empirical data and does not know how much energy a task would have consumed if the situation on the sibling CPUs had been a different one. This introduces errors in the energy profiles of tasks running on SMT processors. We will discuss the implications this has on energy–aware scheduling in Section 3.6.6.

Since scheduling decisions and task switches happen independently for the sibling CPUs of a simultaneously multithreaded processor, the energy profiler maintains power consumption rates separately for each logical CPU. Since the logical CPUs operate in parallel, the power consumption rate of the physical processor is the sum of the consumption rates of all its logical CPUs.

3.6 Energy-Aware Scheduler

3.6.1 Motivation and Objectives

The goal of our energy–aware scheduling policy is to avoid thermal imbalances and high temperatures of individual processors. Section 2.1.1 states that conventional SMP schedulers migrate a task from one CPU to another if there is a load imbalance. However, an energy–aware scheduler may find different reasons for moving tasks than unequal runqueue lengths: For instance, hot tasks might have to leave hot processors, so these processors do not become even hotter and have to be throttled. If this is not done exceedingly often, the penalty incurred by migrating tasks is redeemed by having all processors running at full speed. Nevertheless, we still want the load of the processors balanced, since we want to keep the property of providing fairness to the tasks.

In Section 3.1.1, we identified imbalances in power consumption and non–uniform cooling properties of the processors as the reasons for thermal imbalances. For simplicity, we first assume that all processors possess the same cooling properties. In this case, we can balance the processors' temperatures by balancing their power consumption. We achieve this by combining hot tasks with cool tasks on each CPU in a way that the the average of the power consumption is the same for all runqueues. Once power consumption is balanced, there is no further need for task migrations. However, it is not always possible to balance power consumption exactly, e.g. in a system with many hot tasks and one single cool task. In this case, we have to do migrations from time to time, but since the processors' temperature only change slowly, the number of migrations is still low.

Balancing temperature by balancing power consumption only works if all processors possess the same cooling properties. A processor that is located closer to some cooling facility like a fan or an air intake than another one is able to dissipate more energy per time unit without overheating and can thus endure a higher maximum power. Since there is a linear relationship between (static) power consumption and temperature (see Equation 2.2), we have to balance the power consumption with respect to the maximum power that a processor is able to dissipate without overheating to keep all of a system's processors at the same temperature. Hence, our scheduling policy strives to equalize the ratios between the processors' power consumption and their respective maximum power. This way, we compensate the processors' non–uniform cooling properties by directing power consumption towards the processors with better cooling properties.

For processors running only one task, balancing power consumption by combining tasks with different energy characteristics is not applicable. Hence, we apply a different policy in such situations. The high power consumption of hot tasks causes a processor executing a hot task to heat up. To avoid having to throttle such processors, we migrate hot tasks from hot processors. A hot task can then continue to run on a cooler processor, which has either been idle before, or has been running a cool task. In the second case, the cool task can continue to run on the hot processor. We minimize the number of task movements by we migrating a single hot task at the latest possible moment, when the processor it runs on is about to reach its maximum temperature.

In summery, we pursue two different energy–aware scheduling policies, depending on the number of tasks running on a CPU: In situations with more than one task per CPU, we employ *energy balancing*, a form of load balancing which takes the tasks' energy profiles into account and tries to distribute energy consumption evenly among all CPUs. In situations when there is only one or less runnable task on a CPU, we employ *hot task migration* to move tasks away from processors that threaten to overheat.

3.6.2 Input Values

We will now discuss the input values on which the energy–aware scheduling algorithm's decisions are based. Most of the scheduler's input values are provided by the energy estimator, and have already been introduced in Section 3.5: the task energy profiles as well as the calculated and the empirical power consumption rates.

As motivated in the preceding subsection, we do not necessarily want to balance the consumption rates, but rather their ratios relative to a CPU–specific maximum power that the processor can withstand on the long run without overheating. Therefore, this maximum power, which is a constant depending of the processors thermal characteristics, must be known to the scheduler. If the allowed maximum temperature for a processor, ϑ_{max}, the ambient temperature ϑ_0 and the processor–specific constants c_1 and c_2 are given (see Equation 2.2), we can calculate the maximum power P_{max} for a processor using the static part of Equation 2.2:

$$P_{max} = \frac{c_2}{c_1} \cdot (\vartheta_{max} - \vartheta_0) \tag{3.6}$$

The meaning of this maximum power is: If the power consumption of a processor is greater than P_{max}, on the long run, the processor will overheat. If the power consumption is only temporarily greater than P_{max}, this does not necessarily lead to overheating, since the temperature does not rise immediately, but gradually as described by the dynamic part of Equation 2.2. The empirical consumption rate is calibrated to this dynamic behavior (see Section 3.5.6). Therefore, a processor's temperature is below ϑ_{max} as long as its empirical consumption rate is below P_{max}.

As already mentioned, energy–aware scheduling does not guarantee that the processor temperature is kept below ϑ_{max}. The maximum power is only a hint for the scheduler, who tries to keep power consumption below this rate in a best–effort manner.

From the maximum power and the power consumption rates, we can derive the ratio between the calculated power consumption rate and the maximum power and the ratio between the empirical power consumption rate and the maximum power. So in total, the following input values are available to support the scheduler's decisions:

For each task

- the energy profile

and for each CPU

- the calculated power consumption rate
- the empirical power consumption rate
- the maximum power
- the ratio between calculated power consumption rate and the maximum power
- the ratio between empirical power consumption rate and the maximum power.

3.6.3 Energy Balancing

The energy–aware scheduler applies energy balancing if a runqueue consists of two or more tasks. The goal is to combine the tasks of different runqueues in a way that results in the same power consumption ratio (and therefore the same temperature) for each CPU. For scalability reasons, energy balancing uses a distributed algorithm similar to the load balancing algorithm found in Linux: The algorithm runs on every CPU, possibly in parallel and works in two steps: During the first step, the algorithm searches for another queue to do balancing with. The second step consists in moving tasks between the two queues in order to resolve imbalances. Only tasks which are currently not executing but waiting until it is their turn to get a timeslice of CPU time are transfered this way. This is called *passive balancing*.

As the balancing algorithm is executed periodically on every CPU, balancing needs only be done in one direction: The Linux load balancer, e.g., only pulls in tasks from remote runqueues to resolve imbalances. If there is an imbalance which would require pushing out tasks, this imbalance is resolved when the load balancer runs on the remote CPU. Similarly, energy balancing shall (with some exceptions) be done only by pulling in "heat" from other runqueues. Since we want to balance the processors' temperatures by balancing their power consumption, transferring "heat" from CPU A to CPU B means finding a task whose migration from A to B causes A's calculated power consumption rate to decrease. Therefore, the energy profile of the task in question must be bigger than A's calculated power consumption rate.

Since migrations incur some overhead, we balance rather conservatively to avoid ping–pong effects (tasks being migrated back and forth). Two conditions must be satisfied to justify the migration of a hot task from CPU A to CPU B: The temperature of CPU A must be higher than the temperature of CPU B, and CPU A must be consuming more power with respect to its maximum power than CPU B. Translated to power consumption ratios, this means that A's empirical power consumption ratio must be greater than B's, and A's calculated power consumption ratio must be greater than B's.

This way, we limit the number of task migrations: On one hand, since the empirical power consumption ratio is only changing slowly, this introduces a hysteresis effect; tasks that we moved in one direction stay on the target CPU for some time until the need for moving them back can arise. On the other hand, the calculated power consumption

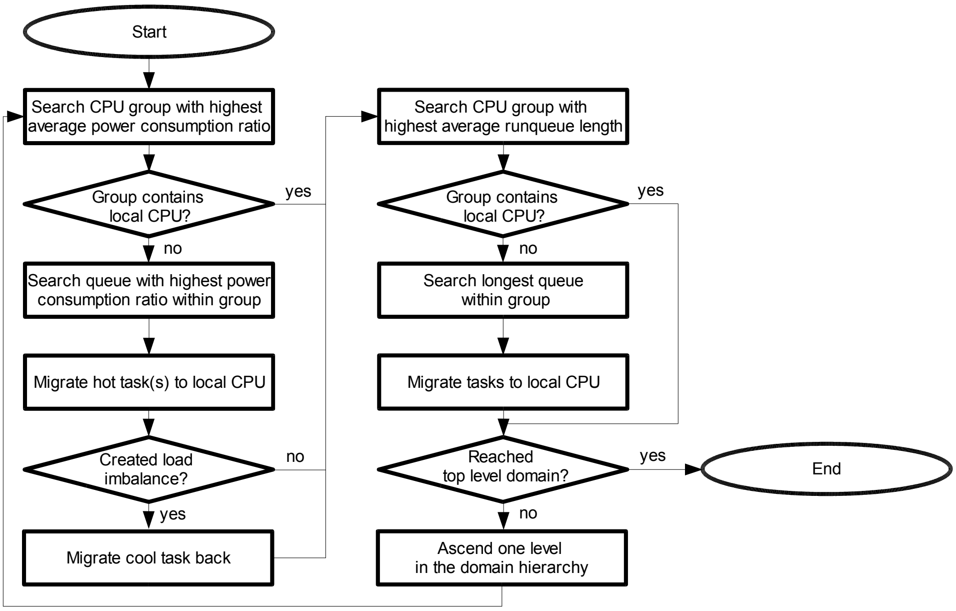

Start
Search CPU group with highest average power consumption ratio
Group contains local CPU?
yes
no
Search queue with highest power consumption ratio within group
Migrate hot task(s) to local CPU
Created load imbalance?
no
yes
Migrate cool task back
Search CPU group with highest average runqueue length
Group contains local CPU?
yes
no
Search longest queue within group
Migrate tasks to local CPU
Reached top level domain?
yes
no
Ascend one level in the domain hierarchy
End

Figure 3.3: Energy and load balancing algorithm

ratio, changing immediately, forbids to migrate an undue number of tasks, so we do not replace an imbalance in one direction with another imbalance in the other direction.

Since we want to keep all runqueues to the same length, we have to apply energy balancing and load balancing. Energy balancing decisions must be consistent with load balancing decisions and vice versa. Otherwise, a task movement made for energy reasons might be undone again for load balancing reasons. Therefore, the energy balancer must always strive not to create load imbalances and the load balancer must strive not to create energy imbalances. Since load and energy balancing are intertwined (energy consumption is always bound to tasks), we decided to merge load balancing with energy balancing into one algorithm.

Figure 3.3 depicts our combined energy and load balancing algorithm. The scheduler respects the domain hierarchy when doing balancing and tries to resolve imbalances on the lowest level possible (see Section 2.1.2). Therefore, the scheduler executes the following steps for each domain a CPU doing balancing belongs to, beginning with the lowermost.

First it searches for the CPU group with the highest average power consumption ratio in order to do energy balancing. If the group containing the local CPU is found to be the one with the highest ratio, which is probable because we are comparing conservatively as described above, there is no need to do energy balancing and the scheduler proceeds directly to the load balancing step. Otherwise, it searches for the queue with the highest ratio among the queues in the remote group to do balancing with this queue. It migrates tasks from the remote to the local queue that have energy profile values higher than the calculated consumption rate of the remote queue in order to cool down the queue. However it does only migrate such a number of tasks that it does not create a load imbalance or an energy imbalance in the wrong direction. Sometimes, it might not be possible at all to migrate tasks without creating a load imbalance, e.g. if both queues are of equal length. In this case, the scheduler migrates a low energy task from the local queue back to the remote queue in exchange.

Then the scheduler searches for the most loaded CPU group in order to do load balancing. If the local CPU belongs to the most loaded group, there is no need to do load balancing. Else the scheduler searches for the longest runqueue in the group concerned. Again, it migrates tasks from the remote to the local queue without creating a load imbalance in the opposite direction and without creating an energy imbalance. Opposed to energy balancing, load balancing can always be done without creating an energy imbalance: If the remote runqueue is hotter than the local one, the scheduler migrates a task that is hotter than the remote queue's consumption rate. If the remote runqueue is cooler, it migrates a task cooler than the remote queue's consumption rate.

3.6.4 Hot Task Migration

As described in Section 3.6.1, we want to move a task to another CPU (assuming there is a suitable one), if it has caused the CPU it is running on nearly to overheat. So if a CPU's empirical consumption rate rises above its maximum power — which, due to the calibration of this rate to the CPU's thermal characteristics, is coterminous with the CPU having reached its temperature limit — and only one task is in the CPU's runqueue, we use *active migration* to transfer the task elsewhere. Since a task cannot be migrated while running, we have to stop it prior to migrating it. Therefore, active migration is done by a special task, which is woken up to preempt the currently running task and to transfer it to its destination CPU.

If there already is one task or more in the runqueue of the destination CPU, simply transferring the hot task would create a load imbalance between the source CPU, which would be idle after the migration, and the destination CPU, which would be running several tasks. Therefore, in such situations, we move a cool task back from the destination CPU to the source CPU in exchange.

The destination CPU, to which the task is transfered, should be a CPU that is significantly cooler than the source CPU. Otherwise, the hot task would soon have to be migrated again. Also, only idle CPUs or CPUs running a cool task are suitable as destination CPUs, since we do not want to migrate a hot task back in exchange. Unless necessary, migrations across node boundaries should be avoided in NUMA systems.

Figure 3.4 depicts the hot task migration algorithm. To keep the cost of the migration as small as possible, the scheduler traverses the domain hierarchy similar to energy balancing bottom–up to find a suitable destination CPU. For each domain level, beginning with the lowermost, the scheduler searches for the the coolest CPU within the local domain. If the difference between the empirical consumption rate and the

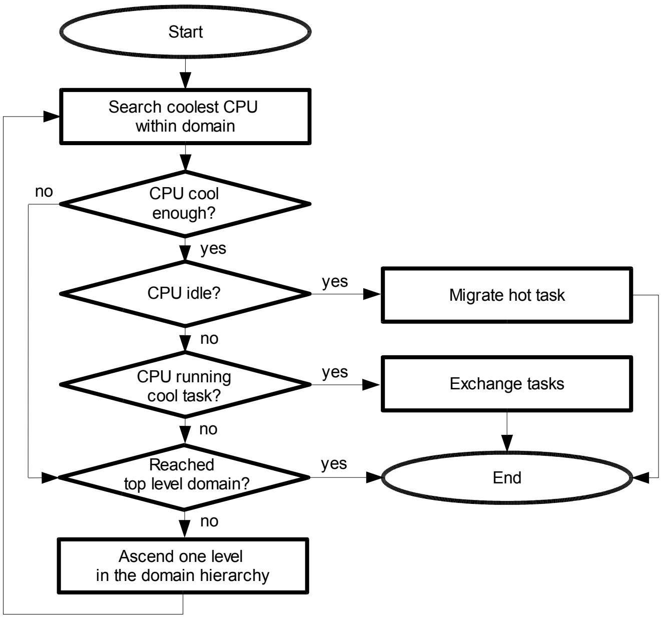


Figure 3.4: Hot task migration algorithm

maximum power of this coolest CPU is bigger than some predefined constant, and the CPU's runqueue is empty or contains a cool task, the scheduler chooses the CPU as the destination, else it continues searching one level higher in the domain hierarchy.

If no suitable CPU is found after searching the top–level domain, all of the system's CPUs are hot and the hot task must remain on the hot CPU, which in turn will have to be throttled.

3.6.5 Initial Task Placement

In any case, whether energy balancing or hot task migration is applicable, choosing the right CPU for a newly started task is important. An inauspicious placement entails the need for migrations in the near future. Section 3.5.4 described how the energy profiler makes a task's energy profile available before the task starts executing. Using this initial energy profile, the scheduler chooses a fitting CPU for the task. First of all, we want to avoid creating load imbalances. Therefore, a CPU is only eligible for running the new task, if there is no other CPU currently running fewer tasks.

For each of the CPUs in question, the scheduler calculates what the ratio of the power consumption rate with respect to the maximum power would be if the new task were assigned to the CPU. Since our goal is to have the same ratio for all CPUs, we assign the new task to the CPU whose ratio, including the new task, comes closest to the average ratio of all CPUs. This way, hot tasks get assigned to cool CPUs and cool tasks get assigned to hot CPUs.

3.6.6 SMT Issues

Since the logical CPUs of a simultaneously multithreaded processor all dissipate their heat on the same physical chip, there is no need to do energy balancing between them. We take care of this by means of the scheduler domain abstraction: We mark the scheduler domains on the lowest level, which encompass all logical CPUs belonging to the same physical processor, with a special flag. This tells the scheduler not to do energy balancing, so it skips the energy balancing step for those domains. Load balancing, on the other hand, must still be done between sibling CPUs, but the energy restrictions for load balancing mentioned in Section 3.6.3 do not apply.

As the logical CPUs of one processor all dissipate their heat on the same chip, we divide the maximum power that a physical processor can endure without overheating between all its logical CPUs. Since we do not do energy balancing between sibling CPUs, it may be that one logical CPU operates above its maximum power, while another one operates below. On the next higher scheduler domain level, where we do energy balancing, all logical CPUs of one processor are collected in one group, and only the average of the group matters. Hence, the scheduler considers energy balanced, when the power consumption of the physical processors are balanced, while there may still be imbalances between the logical CPUs of each processor.

Similarly, since not logical but only physical processors can overheat, we only migrate a hot task actively from a logical CPU belonging to a simultaneously multithreaded processor, if the sum of the empirical consumption rates of all logical CPUs belonging to a physical processor is greater than the allowed maximum power for that processor. Again, we skip the lowermost level of the scheduler domain hierarchy when searching for a destination CPU, since migrating the hot task to a sibling CPU would not improve the situation.

In Section 3.5.7, we mentioned the problem that the energy profile of a task depends on whether the task runs alone on a physical processor or together with other tasks running on sibling CPUs. The energy profile of a task that is migrated from a logical CPU with busy siblings to another chip's logical CPU whose siblings are idle (or the other way round) changes significantly in most cases.

Hot task migration has to deal with this problem, since it is applicable in situations in which not all logical CPUs of the system are busy. It is possible that some tasks are running on logical CPUs with busy siblings and other tasks are running on logical CPUs with idle siblings. We solve the problem by not directly comparing the energy profiles of tasks running on different physical processors, but by comparing the CPU specific consumption rates of the physical processors, which are the sums of the consumption rates of the logical CPUs. This is sufficient, since for hot task migration, we only need to know whether a physical processor is hotter than another physical processor, but need not compare the power consumption of logical CPUs or tasks.

Energy balancing is only applicable when there are several tasks in a runqueue. Since the load balancer ensures that no CPU is idle while another CPU has more than one task in its runqueue, our energy–balancing algorithm only has to deal with tasks that run on logical CPUs having busy siblings and never with tasks running on logical CPUs having idle siblings. Although the problem mentioned above hence does not concern energy balancing, there is a similar problem when migrating tasks between logical CPUs with busy siblings: Even the energy profile of a task that is migrated between logical CPUs with busy siblings might change after the migration, since the power consumption of each task depends on the combination of tasks running on a chip's logical CPUs.

This means that a task migration might not have the desired effect on the power consumption of the processors, since the tasks' energy profiles might change unfavorably after the migration, which entails the need for further migrations. However, with the siblings of source and target CPU busy, the change in the tasks' energy profiles is only minor in most cases. For our test applications (see Chapter 5), the profile varies about 10% on average for different combinations of applications. Hence, energy balancing still works well on SMT processors. Our measurements showed the same benefits of energy balancing for SMT and non–SMT processors.

4 Implementation

We implemented the three components described in the preceding chapter — energy estimator, energy profiler, and energy–aware scheduler — for the Linux kernel. Our modifications done to Linux in order to integrate the components sum up to roughly 2000 lines of C–code. We implemented the energy estimator, which is the only architecture dependent component, for the Intel Pentium 4 processor.

In a symmetric multiprocessor system, every CPU can execute kernel code. Hence, a kernel function can be invoked on different CPUs. Often, the data a function is working on depends on the CPU on which the function is running. For example, `schedule()`, the main scheduler function, works on the runqueue data structure associated with the CPU on which `schedule()` is executing. In this chapter, we use terms like "the scheduler of CPU X" in the meaning of "the function implementing the scheduler running on CPU X and working on CPU X's runqueue".

4.1 Energy Estimator

We extended the energy estimator taken from the resource container implementation by Martin Waitz [Wai03] to support Pentium 4 processors with HyperThreading. A Pentium 4 with HyperThreading is a two–way simultaneously multithreaded Pentium 4 processor. Since we do energy estimation separately for both logical CPUs, each logical CPU runs an energy estimator of its own.

As described in Section 3.4.3, we distinguish three categories of events on the Pentium 4: Events that can be attributed to logical CPUs and can be counted separately, events that can be attributed to logical CPUs and require multiplexing, and events that cannot be attributed to logical CPUs.

Of the eight events chosen for energy estimation (see Table 4.1), three fall into the first category: Writes to the µop queue, retired branches, and mispredicted branches can be counted separately for each logical CPU. Accesses to main memory and first level cache misses require multiplexing, since for these events, there is only one counter register for both CPUs. Load replays from the memory reorder buffer, leakage power, which we model by weighting the time stamp counter, and the base power the processor consumes when not in sleep mode, modeled by counting unhalted cycles, cannot be attributed to logical CPUs.

Simultaneously multithreaded Pentium 4 processors exhibit a peculiarity having to do with consistency. From an architectural point of view, the sibling CPUs of a SMT processor behave like distinct processors: Each sibling has its own set of registers.

event	counting method
µop queue writes	individually
retired branches	individually
mispred branches	individually
mem retired	multiplexed
ld miss 1L retired	multiplexed
time stamp counter	divided
unhalted cycles	divided
mob load replay	divided

Table 4.1: Events used for energy estimation

The event monitoring registers are an exception to this, since they are shared by both siblings. If one sibling modifies an event monitoring register, the change is visible to the other sibling. Therefore, concurrency issues must be considered when sibling processors access the same event monitoring register, as is the case with the registers we multiplex or share between the siblings.

For events which require multiplexing counter registers , we implemented a master–slave approach to solve concurrency the problem. For each multiplexed counter register, one logical CPU is the master. The energy estimator running on this master CPU is responsible for doing the multiplexing. It maintains two counter values in main memory, one for each logical CPU. On every timer tick, it stores the value of the counter register to memory and reloads it with the value saved for the other logical CPU on the previous timer tick. Then the energy estimator reprograms the control register associated with the counter and instructs the counter to count events for the other logical CPU. The energy estimator on the slave logical CPU never deals with the counter register, but only with the counter value residing in memory.

The energy estimator accesses all non–multiplexed counter registers read only. On every invocation, it compares the actual counter values with the counter values of the last invocation in order to determine the energy consumption. This also applies to registers counting events that cannot be attributed to distinct logical CPUs. Although such registers are shared by two logical CPUs and count events occurring on both CPUs, consistency is guaranteed, since the energy estimators on both CPUs access the counter read only. The energy estimator on each logical CPU reads the register, compares its value to the value of the last invocation, and assumes that half of the events that happened in the meantime happened on account of the local logical CPU.

The original implementation of Martin Waitz was designed for sampling energy consumption on every timer tick. This turned out to be insufficient for characterizing tasks with periods of execution shorter than a timer tick. This is e.g. the case for two tasks linked by a pipe, where both tasks block and unblock frequently, waiting for the pipe's buffer to be filled or emptied. Therefore, we modified the implementation to be able to sample energy consumption at arbitrary points in time, so the energy profiler can invoke the estimator whenever a task stops executing.

4.2 Energy Profiler

4.2.1 Task Energy Profiles and CPU Power Consumption Rates

Since a task energy profile is associated with a specific task, we implemented it by incorporating additional fields into the `task_struct` data structure, which Linux uses to describe a task.

The CPU power consumption rates, on the other hand, are CPU–specific. In Linux, the `runqueue` is a CPU–specific data structure, which — besides a list of currently runnable tasks — also contains other information relevant to scheduling. We implemented the empirical and the calculated CPU power consumption rate as two additional fields in the `runqueue` data structure. A further field holds the time constant used for calculating the empirical consumption rate.

4.2.2 Initial Energy Profiles

Whenever the operating system starts a new task, the energy profiler must initialize the task's energy profile. We chose the approach of storing the initial energy profiles, which the energy profiler assigns to newly started tasks, in a global hash table. The management of those initial energy profiles in a hash table is simple since we do not need any collision management. Assuming that collisions occur rarely, it does not matter if a wrong initial energy profile is assigned to a task: The initial energy profile is only a hint for the scheduler regarding the initial placement of a task and is soon replaced with the true profile determined by online energy estimation. The only consequence of a wrong initial profile is a suboptimal placement decision of the scheduler, which can be tolerated from time to time.

In Linux, a file can be uniquely identified by its inode number in combination with the device number of the device hosting the file system. Hence we use the inode and device number of a task's corresponding binary file to calculate the hash key.

Like in many Unix systems, in Linux, the creation of a new task from a binary file image happens in to steps: First, a new task is created with the `fork()` system call, which, simply put, duplicates the task issuing that system call. The new task is called the *child*, whereas the originating task is called the *parent*. The child then issues the `execve()` system call, which replaces the memory image of the child with the binary image loaded from a file.

In many, but not in all cases, a `fork()` system call is followed by an `execve()` call. Some applications just use `fork()` to duplicate themselves and then work with multiple threads of control. Recent versions of Linux offer the `clone()` system call, which allows multithreaded applications a fine grained control over which parts of the parent's runtime image should be duplicated and which ones be shared between the parent and the child.

The energy profiler considers these two possibilities: If a child is created via a `fork()` or a `clone()` system call, it inherits the energy profile of the parent, since it is an identical copy and thus has the same energy characteristics as the parent.

As soon as a task issues an `execve()` system call, the energy profiler replaces the task's energy profile. On an `execve()` system call, the energy profiler calculates a hash key for the file holding the image being loaded, and stores this key in the `task_struct` data structure describing the task. Then, it uses the key to retrieve the task's initial energy profile from the hash table. If there is no entry for the task concerned in the hash table yet, it uses a default value.

If a newly started application finishes its first timeslice, the energy profiler stores the amount of energy consumed during this first timeslice to the hash table using the key saved in the `task_struct` data structure. This way, if the same application is started anew at a later point in time, the energy profiler can use the saved value to initialize the energy profile.

4.2.3 Updating

The energy profiler must update a task's energy profile on two occasions (see Section 3.5.2): at the end of a timeslice and on a task switch. In Linux, the scheduler is the only component that triggers task switches. The scheduler also detects the end of a timeslice; it gets invoked every timer tick to check whether the timeslice of the currently running task is over yet. Hence, we let the scheduler invoke the energy profiler at the end of a timeslice and prior to a task switch. (If the end of a timeslice entails a task switch, we invoke the energy profiler only once.) The energy profiler then updates the energy profile of the task that ran up to the point of invocation as well as the local CPU's empirical power consumption rate. Since the calculated power consumption rate is the average of the energy profiles of all tasks in a CPU's runqueue, the energy profiler has to recalculate this rate, too, after updating the energy profile of the currently running task.

The energy profiler must also update the calculated consumption rate whenever a task is added to a runqueue or leaves a runqueue. Therefore, the scheduler (which is the only component to modify the runqueues) notifies the energy profiler on such occasions.

Linux protects each runqueue with a spinlock from concurrent accesses. Since changes to the power consumption rates occur only on occasions on which the corresponding runqueue is locked (at the end of a timeslice, prior to a task switch, and when a task is added to or removed from the queue), the runqueue's spinlock also provides consistency for the consumption rates. The same applies for the task energy profiles: A task's energy profile is only modified by the energy profiler running on the CPU the task is currently executing on, and only on occasions when the CPU's runqueue is locked.

4.3 Energy–Aware Scheduler

4.3.1 Input Values

As mentioned in Section 4.2.1, the CPU power consumption rates are stored in the runqueue, where the scheduler can access them. In addition to the power consumption rates, the scheduler needs to know the maximum power for each CPU, which we also store in the runqueue.

The scheduler calculates the ratios of the empirical and the calculated power consumption rate with respect to this maximum power on the fly whenever needed, for the consumption rates change rather frequently. Additionally, the scheduler often does not need the current ratio, but has to calculate what the ratios would be like if a specific task were migrated from one CPU to another one.

4.3.2 Energy Balancing

We implemented energy balancing by modifying Linux's load balancer. Linux runs the load balancer periodically for each scheduling domain. The time between two subsequent invocations of the load balancing algorithm is domain–specific and depends on whether a CPU is busy or idle, and on whether previous balancing attempts were successful or unsuccessful. We replaced the original Linux load balancing algorithm with our combined energy–load balancing algorithm described in Section 3.6.3.

In addition to the periodic runs of the balancer, Linux also attempts to do load balancing whenever a task issues an `execve()` system call. `execve()` replaces the task's memory image. Hence, a task has no memory and cache footprint when issuing `execve()`, and can be migrated almost without overhead. Again, instead of doing pure load balancing on `execve()`, we apply the combined policy described in Section 3.6.5 based on the new energy profile the energy profiler assigns to a task on `execve()`.

4.3.3 Hot Task Migration

We extended the kernel thread already present in Linux for doing active migration to support hot task migration. There is one migration thread for every CPU in the system. These threads are inactive when there are no migrations to be done.

To detect the need for hot task migration, we extended the `scheduler_tick()` function, which executes every timer tick on each CPU. In this function, we compare the CPU's empirical consumption rate to the allowed maximum power. If the empirical rate is greater than the maximum power, and the conditions for a hot task migration (the local CPU is running only one task and there is a suitable destination CPU) are given, we instruct the migration thread of the local CPU to do a hot task migration. For this purpose, we added two further fields to the `runqueue` data structure. One holds the task that the migration thread is supposed to migrate, and the other holds the destination CPU. After filling those fields, we wake up the migration thread.

If the destination CPU is already running one or more tasks, we wake up the migration thread of the remote CPU, too, after having filled out the corresponding fields of the remote runqueue, in order to migrate a cool task from the remote CPU to the local one in return. This second active migration is not strictly necessary, since after transferring the hot task, the destination CPU would be running two or more tasks and thus the cool task could be migrated passively. However, passive migration would have to wait until a point in time at which the cool task is currently not executing, and until then, the hot CPU would be idle. Active migration happens immediately, and therefore yields a better utilization of the hot CPU.

4.4 Proc Interface

Linux provides the proc filesystem for exporting information from the kernel to userspace and for letting the user modify certain kernel parameters at runtime. We use the proc–interface to export information related to energy–aware scheduling in which the user might be interested, and to allow the user to calibrate the energy–aware scheduler to the thermal characteristics of the processors.

4.4.1 Task Energy Profiles

Besides its use for energy–aware scheduling, the energy characteristics of a task can also be interesting to the user. Given two different applications that are able to perform the same work, the user might want to know, which one is more efficient in terms of energy consumption. Exporting the task energy profiles to userspace allows users to take advantage of the energy estimation mechanism present in the kernel.

We therefore added the file `/proc/<PID>/energy` to the task–specific part of the proc filesystem. A read from this file delivers the current energy profile of a task.

4.4.2 Energy–Aware Scheduler

We also added the file `/proc/eas` to the proc filesystem. Reading this file reveals the current empirical and calculated power consumption rate as well as the allowed maximum power of each CPU for informational purposes.

Writing to `/proc/eas` allows the user to set the allowed maximum power and the time constant for the empirical power consumption rate.

Using this interface, the user (or an automated userspace program that has access to temperature values from the thermal diode present in almost all modern systems to measure the processor temperature) can calibrate the energy–aware scheduler to the thermal characteristics of each CPU.

4.5 Throttling

We also implemented a crude throttling mechanism to demonstrate how energy–aware scheduling mitigates the need for throttling. We throttle a processor by executing the `hlt` instruction, which puts the processor in a low power sleep state until the next interrupt arrives. We trigger this mechanism whenever a CPU's empirical consumption rate rises above the maximum power. Because of the calibration of the empirical consumption rate to the thermal characteristics of the processor, this is coterminous with the processor's temperature rising above the limit temperature.

Although this mechanism is based on measures taken from the energy–aware scheduler, it is not part of our energy–aware scheduling framework; energy–aware scheduling is compatible with any throttling mechanism triggered by processor temperature.

5 Experimental Results

We evaluated our implementation with a set of different workloads to show how energy–aware scheduling improves the performance of a system by reducing the need for throttling.

We quantify the benefits of energy–aware scheduling by combining it with a temperature control mechanism that works by throttling hot processors. Using scenarios with long running vs. short running tasks, homogeneous vs. heterogeneous energy characteristics, and many vs. few tasks, we evaluate the influence of the workload on the effectiveness of energy–aware scheduling.

5.1 Test Setup

We ran our modified Linux kernel on an 8–way Pentium 4 Xeon multiprocessor ($2.2GHz$ each processor) consisting of two NUMA nodes with four processors each. The processors offer HyperThreading. We ran some of the tests with HyperThreading enabled and some with HyperThreading disabled to test the impact of simultaneous multithreading on energy estimation and energy–aware scheduling.

Table 5.1 summarizes the programs we used in our tests. Most of the programs show static energy characteristics, with the exception of OpenSSL, which we ran in benchmark mode. Because of the different encryption and checksum algorithms benchmarked sequentially, the energy profile of OpenSSL varies between $42W$ and $57W$ over time.

We did evaluations of our energy–aware scheduler with two different classes of workloads. For workloads consisting of long–running tasks (tasks running for several minutes up to hours), task migrations are especially important, whereas for workloads consisting of short–running tasks (tasks running only for some seconds), the initial placement of newly started tasks is most essential. Therefore, two versions of each of the test programs exist: a long running one, which takes several minutes to execute, and a short running one, executing for less than a second.

We ran all programs in an endless loop from a shell script to have a constant load during the whole test run: Whenever one instance of a program terminates, the script starts a new instance of the same program.

program	power	description
`bitcnts`	$61W$	bit counting operations
`memrw`	$38W$	memory reads/writes
`aluadd`	$50W$	integer additions
`pushpop`	$47W$	stack push/pop
`openssl`	$42W - 57W$	OpenSSL benchmark
`bzip2`	$48W$	file compression
`gcc`	$47W$	gcc compiling the Linux kernel

Table 5.1: Programs used for the tests

5.2 Energy Balancing

We ran a workload consisting of the six long–running programs to demonstrate the influence of energy balancing on the processors' power consumptions. We started each program three times, for a total of 18 running tasks. We ran three series of the test:

1. Energy balancing disabled.

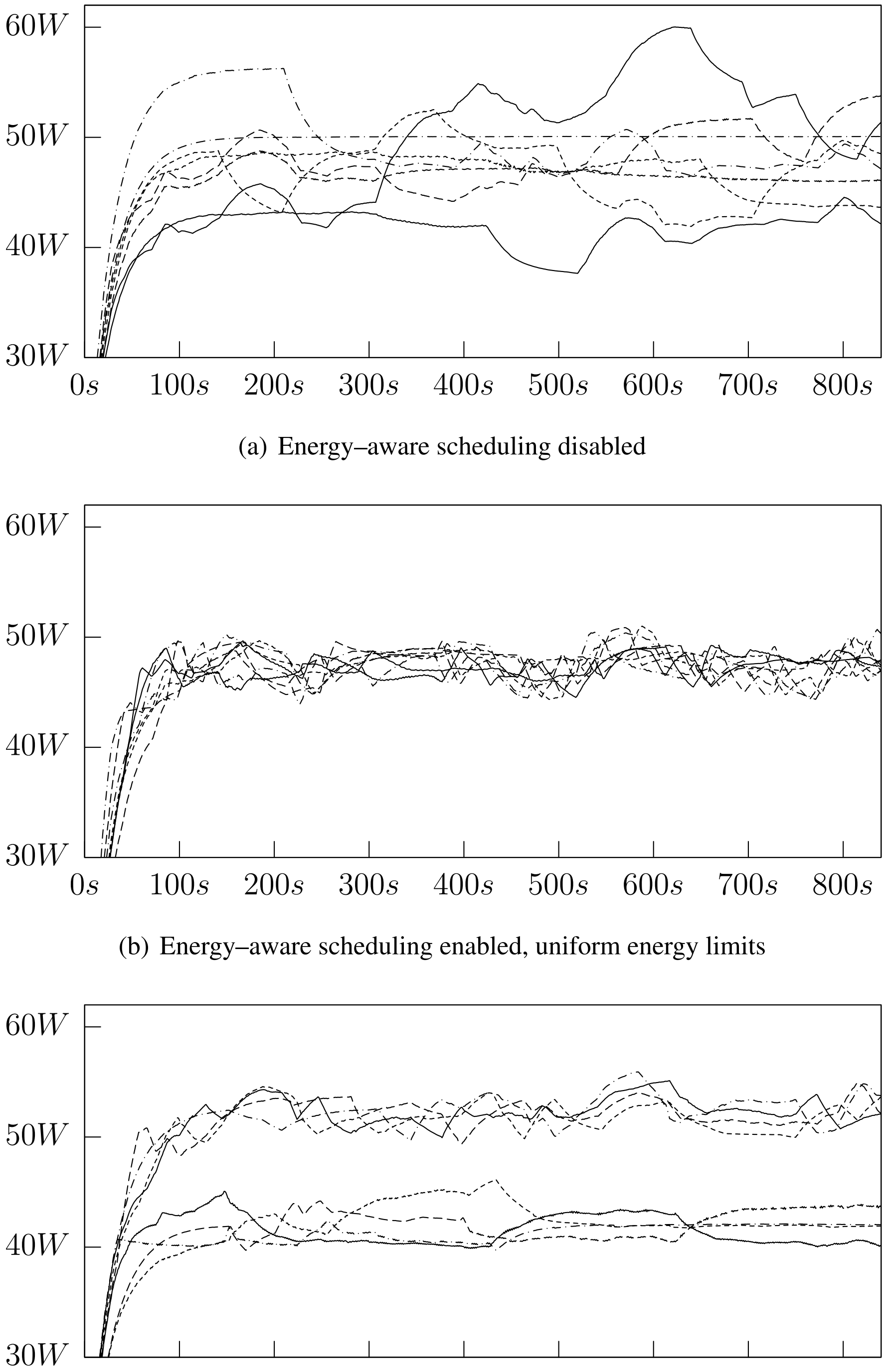
2. Energy balancing enabled with a $60W$ limit for all processors, so so power consumption should be balanced evenly across all processors.

3. Energy balancing enabled with a $40W$ limit for CPUs 0 to 3 and a $60W$ limit for CPUs 4 to 7.

For comparison, we ran the test series first with simultaneous multithreading disabled and then with simultaneous multithreading enabled.

Figure 5.2 depicts the empirical consumption rates of the eight CPUs with energy balancing disabled and with energy balancing enabled (SMT disabled). Because of the exponential average used for calculating these rates, the rates rise exponentially first. This mirrors the rise of the processors' temperatures. With energy balancing disabled, the curves diverge during the further course of the experiment because of the different energy characteristics of the tasks running on each CPU (Subfigure (a)). With energy balancing enabled and uniform energy limits, the curves stay close to each other, since the energy–aware scheduling policy triggers task migrations whenever the consumption rates of two CPUs differ too much (Subfigure (b)). With non–uniform energy limits, the scheduling policy migrates low–power tasks to the CPUs with low energy limits and high–power tasks to the CPUs with high energy limits (Subfigure (c)).

We ran the experiments several times for 15 minutes, and counted the number of task migrations during the experiments. On average, there were 3.3 migrations with energy balancing disabled and 32 migrations with energy balancing enabled. Although this is an increase by a factor of nearly ten, the overhead of additional migrations is small compared to the benefit of avoiding the throttling of processors. Considering the total



(a) Energy–aware scheduling disabled

(b) Energy–aware scheduling enabled, uniform energy limits

(c) Energy–aware scheduling enabled, non-uniform energy limits

Figure 5.1: Empirical power consumption rates (long running tasks)

of 18 running tasks we had in our experiment, 32 migrations means that on average each task was migrated less than twice during the 15 minutes, so the overhead is negligible.

We did the same experiments with SMT enabled to test the impact of simultaneous multithreading on energy balancing. Since with SMT, there are 16 logical CPUs instead of 8, we started each program six times, for a total of 36 tasks. The results are similar to those of the experiments with SMT disabled. On average, there are 9.8 migrations with energy balancing disabled and 87 migrations with energy balancing enabled.

5.3 Initial Task Placement

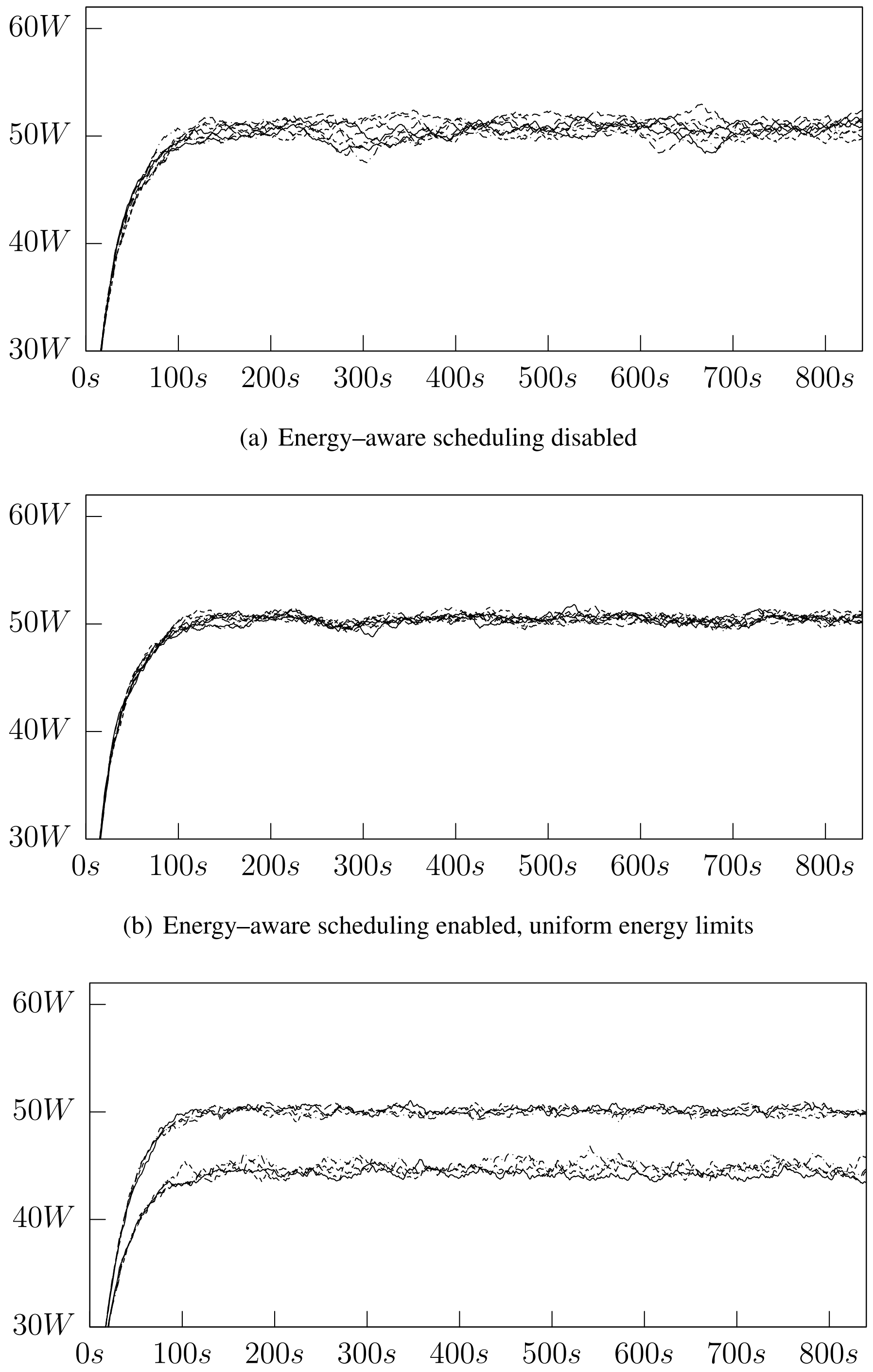
We ran the same test series with a workload consisting of short running tasks to assess the importance of initial task placement. Figure 5.3 depicts the empirical power consumption rates of the CPUs running this workload.

The short runtime of the tasks results in a high frequency at which the tasks are restarted. With energy–aware task placement disabled, the initial placement of the tasks depends solely on the load of the CPUs. From an energy point of view, the placement happens at random. In combination with the high frequency at which tasks are started, this results in power consumption being more or less balanced across all CPUs even with energy–aware scheduling disabled (Subfigure (a)).

If we have uniform energy limits for all CPUs, the benefits of energy–aware task placement are only small (Subfigure (b)). Placing tasks depending on their energy profile results in a slightly smaller width of the array of curves than random placement. In contrast to this, with non–uniform energy limits, considering the tasks' energy profiles for initial task placement results in significantly different power consumptions for the CPUs with high respectively low energy limits (Subfigure (c)). So in this case, the benefit compared to energy–oblivious placement is large.

5.4 Temperature Control

We applied our energy–aware scheduling policy in combination with temperature control by means of throttling to be able to quantify the benefits of energy–aware scheduling. Because of the overprovisioned cooling facilities of our test system, even without energy–aware scheduling and without temperature control, the highest CPU temperature we measured with our mixed workload was only 45°C. Therefore, we set the maximum power of each one of the eight processors in our system to a value that corresponds to a temperature of 38°C. This is not a critical temperature; we only chose such a low value to create a need for throttling at all. As described in Section 4.5, we enforce this temperature limit by executing the `hlt` instruction on CPUs whose empirical consumption rate exceeds the maximum rate.



(a) Energy–aware scheduling disabled

(b) Energy–aware scheduling enabled, uniform energy limits

(c) Energy–aware scheduling enabled, non-uniform energy limits

Figure 5.2: Empirical power consumption rates (short running tasks)

CPU	P_{max}	T
0 (8)	$34W$	$64.9s$
1 (9)	$56W$	$28.8s$
2 (10)	$66W$	$39.0s$
3 (11)	$33W$	$75.0s$
4 (12)	$35W$	$54.8s$
5 (13)	$49W$	$36.1s$
6 (14)	$76W$	$28.8s$
7 (15)	$40W$	$53.4s$

Table 5.2: Thermal properties of our test system

5.4.1 Calibration

We calibrated the energy–aware scheduler to the thermal behavior of our system and to a maximum temperature of 38°C. Therefore, we determined the constants c_1 and c_2 as described in Section 2.2.2. Using Equations 3.5 and 3.6, we calculated the maximum power P_{max} and the super period length T for each processor.

Table 5.2 lists the values we obtained for our test system. The processors of our system possess non–uniformly distributed thermal properties: CPUs 1, 2, 5 and 6 are located closer to the air intake than CPUs 0, 3, 4 and 7 and thus can withstand higher power consumptions without exceeding the temperature limit of 38°C, and cool down quicker when the power changes.

The CPU–IDs shown in brackets represent the IDs of the logical siblings when HyperThreading is enabled. The CPU–IDs of two siblings differ in the most significant bit. As explained in Section 3.6.6, we divide the maximum power of the processor between its logical CPUs in this case.

5.4.2 Benefits of Energy–Aware Scheduling

Since energy–aware scheduling strives to assign tasks to CPUs in a way that keeps the consumption rates for each CPU below the maximum, we expect the need for throttling to be considerably lower if we enable energy–aware scheduling, which will result in more CPU cycles available for executing tasks and thus in a higher throughput. Again, we used the mixed workload listed in Table 5.1 and ran a test with the long running and one with the short running versions of the programs. We obtained the results we are presenting in this section with simultaneous multithreading enabled. Without simultaneous multithreading, the results look similar.

Table 5.3 shows the percentages of the time the CPUs had to be throttled to enforce the temperature limit. In both scenarios, some CPUs have to be throttled neither with energy–aware scheduling disabled nor with energy–aware scheduling enabled because

CPU	long running tasks		short running tasks	
	EAS disabled	**EAS enabled**	**EAS disabled**	**EAS enabled**
0	51.5%	35.1%	61.2%	45.4%
1	0.0%	0.0%	0.0%	0.0%
2	0.0%	0.0%	0.0%	0.0%
3	54.1%	39.7%	49.5%	45.6%
4	10.8%	0.0%	5.0%	0.1%
5	0.0%	0.0%	0.0%	0.0%
6	0.0%	0.0%	0.0%	0.0%
7	0.0%	0.0%	0.0%	0.0%
8	61.1%	35.7%	49.9%	45.7%
9	0.0%	0.0%	0.0%	0.0%
10	0.0%	0.0%	0.0%	0.0%
11	54.7%	51.9%	53.3%	36.9%
12	11.0%	0.0%	46.5%	14.0%
13	0.0%	0.0%	0.0%	0.0%
14	0.0%	0.0%	0.0%	0.0%
15	0.0%	0.0%	9.3%	0.4%
average	**15.2%**	**10.2%**	**18.4%**	**12.7%**

Table 5.3: CPU throttling percentage

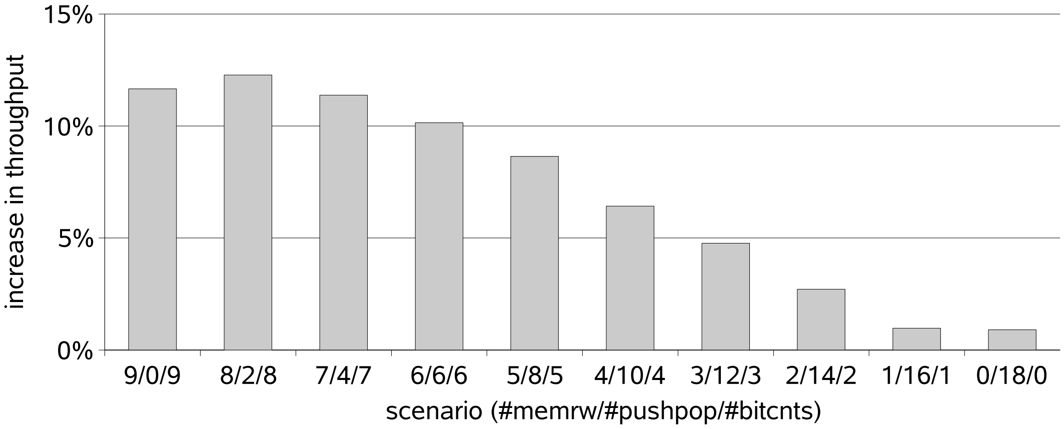


Figure 5.3: Dependence of throughput from workload

of their good thermal properties. For all other CPUs, the throttling percentage is considerably lower when energy–aware scheduling is enabled both for the scenario with the long running tasks as well as for the scenario with the short running tasks. This results in an increase in throughput, which we measured by counting the number of tasks that were finished per time unit. With energy–aware scheduling enabled, the throughput is 4.7% higher for the long running tasks and 4.9% higher for the short running tasks.

5.4.3 Dependence from the Workload

The actual increase in throughput depends on the workload, or more precisely, on how different the energy characteristics of the tasks are. The more heterogeneous a workload is in terms of energy profiles, the more room has the energy–aware scheduler for directing the CPUs' power consumptions by assigning tasks to CPUs. We measured how much energy–aware scheduling increases the throughput for workloads of different homogeneity (Figure 5.3). We created workloads from three applications: bitcnts with a high power consumption, pushpop with a medium power consumption, and memrw with a low power consumption. We ran the tests with HyperThreading disabled and used workloads consisting of 18 running tasks. We started with an inhomogeneous workload consisting of nine instances of memrw and nine instances of bitcnts. For the following tests, we successively replaced one instance of memrw and one instance of bitcnts with an instance of pushpop, until we arrived at a totally homogeneous workload consisting of 18 instances of pushpop.

As we expected, the increase in throughput is best with inhomogeneous workloads, since the energy–aware scheduler can adapt the processors' power consumptions to their thermal properties and run the hot tasks on the processors with better thermal properties and the cool tasks on processors with worse thermal properties. Energy–aware scheduling achieves the highest increase in throughput (12.3%) for the workload consisting of eight instances of memrw and bitcnts, and two instances of pushpop. The increase for this workload is even higher than the one for the workload consisting only of memrw and bitcnts, since some processors in our system possess medium thermal properties, and energy–aware scheduling can therefore best balance temperature when there are some tasks with medium power consumptions to run on these processors. With homogeneous workloads, where all tasks possess the same energy characteristics, energy–aware scheduling has almost no benefit, since the scheduler has no chance for influencing the processors' power consumptions by task migrations.

5.5 Hot Task Migration

Up to now, all the results we presented were obtained with workloads consisting of many tasks that kept all CPUs busy. In this section, we present evaluations done with workloads that leave some CPUs idle. In this case, our energy–aware scheduler applies the hot task migration policy described in Section 3.6.4.

For the first test, we used a single instance of the bitcnts program, consuming about $60W$, for evaluating hot task migration. We chose to use uniform energy limits, which do not correspond to the real thermal characteristics of our system. When the energy–aware scheduler is programmed with the values listed in Table 5.2, like in the previous test, the bitcnts task gets migrated one time at most, since hot task migration transfers it to CPU 2 or CPU 6. Since those CPUs can run bitcnts without overheating, there is no further need for hot task migrations.

Hence, we simulated a system where all processors possess the same thermal characteristics. We allowed each physical processor to consume $40W$ at most to create the need for continuous hot task migrations.

We activated simultaneous multithreading for the test. Since we have only one running task, the sibling of the logical CPU that bitcnts is running on is always idle. If bitcnts is started on or migrated to a CPU formerly idle, it takes approximately ten seconds for the sum of the experimental consumption rates of the two sibling CPUs (one idle, one executing bitcnts) to rise above $40W$. The bitcnts task is then migrated elsewhere by the hot task migration mechanism (Figure 5.4).

Two things are worth noting: Firstly, bitcnts is never migrated to a sibling CPU on the same physical processor. (As already mentioned, the CPU IDs of two sibling CPUs differ in the most significant bit.) Secondly, bitcnts is never migrated across the node boundary: CPUs 0 to 3 (with their siblings 8 to 11) reside on node 0, whereas CPUs 4 to 7 (with their siblings 12 to 15) reside on node 1. The bitcnts task visits the physical CPUs of node 0 nearly in round robin fashion, because the CPU least recently visited is always the coolest one. However, after bitcnts has taken one full turn, the CPU on which it executed first has cooled down enough to avoid inter–node migration.

If bitcnts were executed on one processor all of the time, as is the case without hot task migration, this processor would have to be throttled 33% of the time to enforce the $40W$ limit, assuming that a processor is consuming no energy when throttled. How-

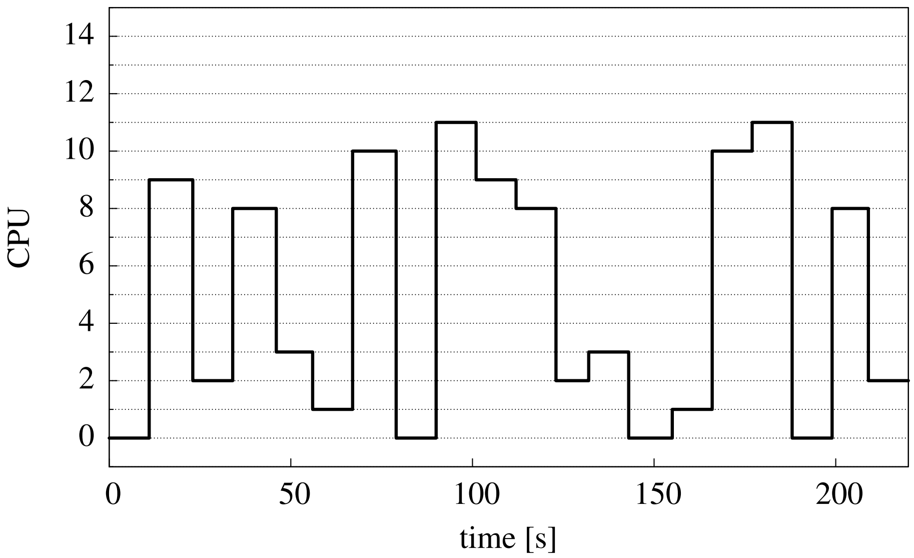


Figure 5.4: Hot task migration of a single task

ever, the processors in our test system consume $13.6W$ when put into a sleep state by executing the `hlt` instruction. Therefore, we even measured a 43% decrease in execution time with hot task migration. This corresponds to an increase in throughput of 76%. Even if we set the maximum power of the processors to $50W$, hot task migration still results in a 21% reducement of the execution time (27% increase in throughput).

For this scenario with only one running task, energy–aware scheduling yields a much higher increase in throughput compared to the scenarios of the previous section. The reason for this is that we always have a cool idle CPU, whither we can migrate the hot task, so we can completely get rid of the need for throttling. Idle processors consume considerably less power than even processors running cool tasks. Therefore, a system with some idle CPUs tends to show greater thermal imbalances which can be taken advantage of by energy–aware scheduling than a system with all CPUs busy.

When a hot processor becomes idle because a hot task has been migrated to another processor, it takes some time for the processor to cool down. Therefore, if there are multiple tasks running in the system, there might not always be a cool CPU available, so the hot task has to stay on the hot CPU, which in turn will be throttled. Figure 5.5 shows the throughput of the system running multiple instances of the bitcnts program relative to the throughput of a system with energy–aware scheduling disabled. With two running tasks, energy–aware scheduling yields the same increase in throughput as with only one task. With two tasks, the first one runs on CPUs 0 to 3 in turns like in the scenario with one task, whereas the second one runs on CPUs 4 to 7. If there are more than two tasks, there are situations when there is no cool target processor because the processors do not cool down fast enough, and some of the time, tasks have to run on

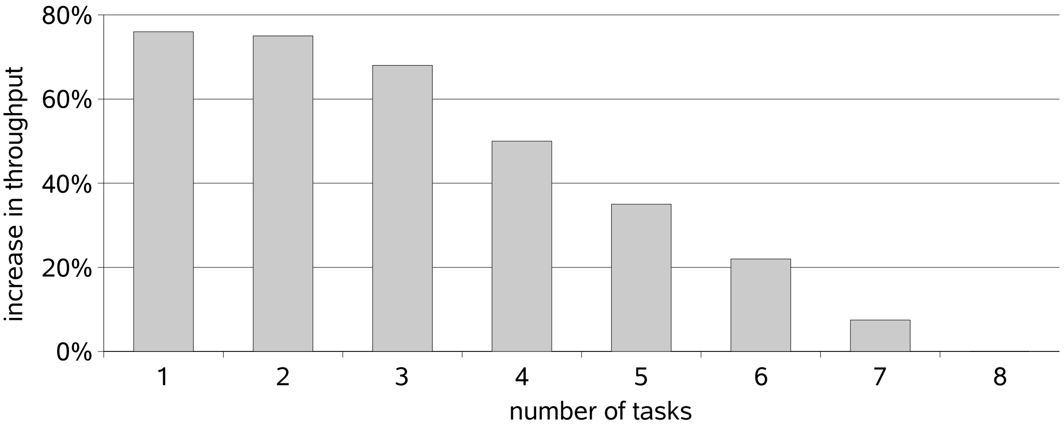


Figure 5.5: Hot task migration — throughput with multiple tasks

throttled processors. Therefore, the more tasks are running, the smaller is the increase in throughput. With eight (or more) running tasks, after some time all physical processors are hot, and there is never a target CPU suitable for hot task migration. This results in a throughput identical to the one of a system without energy–aware scheduling.

6 Conclusions

6.1 Achievements

In this thesis, we showed that the characterization of individual tasks by their power consumption, determined by on–line analysis using event monitoring counters, can be used to influence the power consumptions and temperatures of the processors in a SMP system via scheduling decisions. By employing an energy–aware scheduling policy, the operating system is able to reduce thermal imbalances between the system's processors and thus to mitigate the need for throttling processors, so more cycles are available to userspace tasks. This results in increased performance. Therefore, the two main contributions of this work are

- Task energy profiles as a means for characterizing individual tasks on–line by their power consumption
- Energy–aware scheduling as a means for balancing the processors' power consumptions and for reducing thermal imbalances.

Furthermore, we showed that energy estimation using event monitoring counters can be applied independently for the logical threads of a simultaneously multithreaded processor, provided that the event counters are able to attribute events to the logical CPU that caused them. This way, we were able to individually characterize tasks running quasi–parallel on a SMT processor. With the characterization of individual tasks, we were able to apply energy–aware scheduling to systems with SMT processors using hierarchical balancing.

6.2 Summary

On modern processors like the Pentium 4, different tasks cause different power consumptions depending on the instructions they execute. Processor temperature is dependent on power consumption. In a multiprocessor system, the different processors therefore are likely to have different temperatures. Throttling is a way to prevent hot processors from overheating, but unfortunately, throttling decreases performance. The approach we presented in this work takes advantage of the different energy characteristics of individual tasks and of the thermal imbalances between the processors in a SMP system. We mitigate the need for throttling by assigning tasks to CPUs depending on the tasks' energy characteristics and on the processors' temperatures.

For this, we propose to enhance the operating system by a mechanism for determining the individual tasks' energy characteristics, and by an energy–aware scheduling policy for the operating system's task scheduler. We determine the energy characteristics of a task by means of online energy estimation using event monitoring counters. Therefore, we add a component called energy estimator to the operating system. This component transforms the values from the CPUs' event counter registers to energy values. For representing each individual task's energy characteristic, we introduce task energy profiles. We add a component called energy profiler to the operating system, which maintains these profiles and provides them to the scheduler. The energy profiler also maintains CPU-specific power consumption rates to provide estimates for each processor's power consumption and temperature to the scheduler.

Energy–aware scheduling works by placing new tasks on a suitable CPU and by migrating existing tasks depending on their energy profile. We propose two different energy–aware scheduling policies, energy balancing and hot task migration. We apply energy balancing in situations with multiple tasks running on each CPU. In those situations, we strive to balance the processors' power consumptions and thus their temperatures, so no single processor overheats and has to be throttled, while there are other, cooler processors. We achieve this by combining tasks with different power consumptions on the CPUs. In situations with only one task running on a CPU, there is no chance for balancing power by combining different tasks. Therefore, in such situations, we apply hot task migration: We keep a hot task running on a CPU until the processor chip reaches a critical temperature and then migrate the hot task to a cooler CPU.

Together, the three components energy estimator, energy profiler and energy–aware scheduler constitute an energy–aware scheduling framework, which we implemented for the Linux kernel.

We tested our implementation on an 8 processor Pentium 4 Xeon system with different workloads. Our experiments show that the overhead incurred by the migrations that are required for balancing the processors' power consumptions and temperatures is negligible. On the other hand, energy–aware scheduling reduces the number of cycles during which CPUs have to be throttled by trying to keep all processors' temperatures below the maximum temperature. This way, more cycles are available for executing user programs, which results in increased throughput.

How big the increase actually is, depends on the variety of the energy characteristics of the tasks that the system executes. For typical scenarios with mixed workloads, the throughput increased by about 5% when we enabled energy–aware scheduling. The corner cases are workloads extremely homogeneous in terms of power consumption, where all tasks possess the same energy characteristics, and extremely heterogeneous workloads with some tasks having a very high and other tasks having a very low power consumption. In the first case, energy–aware scheduling has no benefits, whereas in the second case, energy–aware scheduling has maximum benefits.

A special case of an extremely heterogeneous workload is a workload which does not utilize all of a system's CPUs, because the power consumption of idle CPUs is extremely low. Migrating tasks from hot CPUs to idle CPUs instead of throttling the hot CPUs increases performance substantially.

6.3 Future Directions

The biggest limitation of energy–aware scheduling has already been stated in the preceding section: It is only applicable for workloads consisting of tasks with different power consumptions. If all tasks have the same power consumption, there is no need to do energy balancing, since power consumption is inherently balanced.

Currently, most processors are equipped with a single thermal diode. Throttling mechanisms are engaged by some monitoring hardware or software when the temperature value reported by this diode exceeds a certain threshold. Since energy is dissipated at the individual functional units of a processor, the temperature of the chip may be distributed non–uniformly, so decisions about throttling should be based on multiple temperature values. As a consequence, the goal of an energy–aware scheduling policy should be to keep the temperature of all functional units on all processors below the throttling threshold. Future work on energy–aware scheduling could incorporate a more elaborate thermal model featuring multiple temperatures per chip and could characterize tasks not only by their power consumption, but also by the location at which the power is dissipated. This way, energy–aware scheduling would even be beneficial for tasks having the same power consumption, if they dissipate the power at different functional units, as is the case with floating point and integer applications.

We believe that energy–aware scheduling is also applicable to chip multiprocessors (CMP), which will probably become common in future systems. Migrating tasks between individual cores on a CMP for energy reasons is beneficial because different cores on the same chip can have different temperatures. Since our energy–aware scheduling framework uses the scheduler domains abstraction, extending energy–aware scheduling for use on a CMP is a matter of adding an additional layer to the domain hierarchy.

Future multiprocessor systems might consist of CPUs that execute applications at different speeds, e.g. a CMP with fast cores and slow cores, or offer dynamic frequency and voltage scaling for each individual CPU in the system. In such an asymmetric system, migrating tasks between CPUs running at different speeds has an impact on performance and on power consumption. Additionally, the energy characteristics of a task depend on the CPU the task is executing on. This offers new opportunities and new challenges for energy–aware scheduling.

A Appendix

A.1 Calibrating the Power Consumption Rate to the Thermal Model

The empirical power consumption rate models a processor's temperature and must therefore be calibrated to the thermal behavior of the processor, which we model with Equation 2.2.

We assume that the processor has been operating at a static power P_{old} for a long time and has the temperature $\vartheta\left(0\right) = \vartheta_{old}$. Then the power changes to P_{new} at the time $t = 0$. Now the processor's temperature approaches exponentially to the temperature ϑ_{new}.

We describe the situation using our thermal model:

$$\vartheta\left(0\right) = \frac{c_1}{c_2} \cdot P_{old} + \vartheta_0 \tag{A.1}$$

$$\vartheta\left(t\right) = \frac{-\tilde{c}}{c_2} \cdot e^{-c_2 t} + \frac{c_1}{c_2} \cdot P_{new} + \vartheta_0 \tag{A.2}$$

Combining Equations A.1 and A.2, we can eliminate the integration constant $\tilde{c}$:

$$\vartheta\left(t\right) = \frac{c_1}{c_2} \cdot \left(P_{old} - P_{new}\right) \cdot e^{-c_2 t} + \frac{c_1}{c_2} \cdot P_{new} + \vartheta_0 \tag{A.3}$$

Transforming the equation yields:

$$\frac{c_2}{c_1} \cdot \left(\vartheta\left(t\right) - \vartheta_0\right) = \left(P_{old} - P_{new}\right) \cdot e^{-c_2 t} + P_{new}$$

$$\hat{P}\left(t\right) = \left(P_{old} - P_{new}\right) \cdot e^{-c_2 t} + P_{new} \tag{A.4}$$

The left hand side of Equation A.4 is what we want to model with the empirical power consumption rate. It has the dimension of a power, but the dynamic behavior of temperature.

We use an exponential average to calculate the empirical power consumption rate. We assume that all timeslices have the standard timeslice length τ. When operating at the power P_{old}, the processor consumes the energy $v_{old} = P_{old} \cdot \tau$ during a timeslice of length τ; when operating at P_{new}, it consumes $v_{new} = P_{new} \cdot \tau$. We assume that the power rises from P_{old} to P_{new} after timeslice $n = 0$. So the exponential average (compare Equation 3.4) is:

$$\begin{aligned} x_0 &= v_{old} \\ x_n &= \frac{\tau}{T} \cdot v_{new} + \frac{T-\tau}{T} \cdot x_{n-1} \end{aligned} \tag{A.5}$$

To calibrate the exponential average to the thermal behavior of the processor, we have to find a super period length T which corresponds to the time constant c_2 in Equation A.4, so that the exponential average x_n/τ after timeslice n is identical to the exponential function $\hat{P}\left(t\right)$ at the time $t = n\tau$. Transforming Equation A.5 into a non–recursive form yields:

$$\begin{aligned} x_n &= \frac{\tau}{T} \cdot \sum_{i=0}^{n-1} \left(\frac{T-\tau}{T}\right)^i \cdot v_{new} + \left(\frac{T-\tau}{T}\right)^n v_{old} \\ x_n &= \left(1 - \left(\frac{T-\tau}{T}\right)^n\right) \cdot v_{new} + \left(\frac{T-\tau}{T}\right)^n v_{old} \\ x_n &= \left(v_{old} - v_{new}\right) \cdot \left(\frac{T-\tau}{T}\right)^n + v_{new} \end{aligned} \tag{A.6}$$

We divide Equation A.6 by τ:

$$\begin{aligned} \frac{x_n}{\tau} &= \left(\frac{v_{old}}{\tau} - \frac{v_{new}}{\tau}\right) \cdot \left(\frac{T-\tau}{T}\right)^n + \frac{v_{new}}{\tau} \\ \frac{x_n}{\tau} &= \left(P_{old} - P_{new}\right) \cdot \left(\frac{T-\tau}{T}\right)^n + P_{new} \end{aligned} \tag{A.7}$$

We replace t in Equation A.4 with $n\tau$:

$$\hat{P}\left(t = n\tau\right) = \left(P_{old} - P_{new}\right) \cdot e^{-c_2 n\tau} + P_{new} \tag{A.8}$$

Comparison with Equation A.7 yields:

$$\frac{T-\tau}{T} = e^{-c_2\tau} \tag{A.9}$$

This resolves to:

$$T = \frac{\tau}{1 - e^{c_2\tau}} \tag{A.10}$$

List of Figures

List of Tables

Bibliography

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI'99)*, February 1999.

[Bel01] Frank Bellosa. The case for event–driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural–level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, 2000.

[BWWK03] 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)*, September 2003.

[CDV$^+$94] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[DM05] James Donald and Margaret Martonosi. Leveraging simultaneous multithreading for adaptive thermal control. In *Proceedings of the Second Workshop on Temperature–Aware Computer Systems (TACS'05)*, June 2005.

[FM02] Krisztián Flautner and Trevor Mudge. Vertigo: Automatic performance–setting for linux. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, December 2002.

[FS99] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99)*, February 1999.

[GCW95] Kinshuk Govil, Edwin Chan, and Hal Wassermann. Comparing algorithms for dynamic speed–setting of a low–power CPU. In *Proceedings of the first Conference on Mobile Computing and Networking (MOBICOM'95)*, March 1995. Also as technical report TR–95–017, ICSI Berkeley, April 1995.

[GPV04] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat–and–run: leveraging SMT and CMP to manage power density through the operating system. *SIGARCH Comput. Archit. News*, 32(5):260–270, 2004.

[HBA03] Seongmoo Heo, Kenneth Barr, and Krste Asanovi. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED'03)*, August 2003.

[HGS$^+$04] Wei Huang, Shougata Ghosh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R. Stan. Compact thermal modeling for temperature–aware design. In *Proceedings of the 41st ACM/IEEE Design Automation Conference (DAC)*, June 2004.

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high–end processors: Methodology and empirical data. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2003.

[Int03] Intel Corporation. *Event Counters on Intel Pentium 4 Processors, Including HyperThreading Technology Enabled Processors.* 2003.

[Kel03] Simon Kellner. Event–driven temperature control in operating systems. Study Thesis, Operating System Group, University of Erlangen, Germany, April 2003.

[Lin05] `linux/Documentation/sched-domains.txt`. Documentation shipped with the Linux source code, 2005.

[LS05] Kyeong-Jae Lee and Kevin Skadron. Using performance counters for runtime temperature sensing in high–performance processors. In *Proceedings of the Workshop on High–Performance, Power–Aware Computing (HP–PAC)*, April 2005.

[PBB98] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low–Power Electronics and Design (ISLPED'98)*, June 1998.

[RS99] Erven Rohou and Michael D. Smith. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback–Directed Optimization*, November 1999.

[SL93] Mark S. Squillante and Edward D. Lazowska. Using processor–cache affinity information in shared–memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[SSH$^+$03] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature–aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*, June 2003.

[TTG95] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache–affinity scheduling in shared–memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[Wai03] Martin Waitz. Accounting and control of power consumption in energy–aware operating systems. Diploma Thesis, Operating System Group, University of Erlangen, Germany, January 2003.

[WB02] Andreas Weissel and Frank Bellosa. Process cruise control — event–driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'02)*, October 2002.

[WB04] Andreas Weissel and Frank Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the First Workshop on Temperature–Aware Computer Systems (TACS'04)*, June 2004.