Diploma Thesis

Hardware-Based Energy Accounting for Multi-Core Systems

Till Smejkal

March 16, 2016

TU Dresden
Faculty of Computer Science
Institute of Systems Architecture
Operating Systems Group

Supervising Professor: Prof. Dr. rer. nat. Hermann Härtig
Supervising Staff: Dr.-Ing. Michael Roitzsch
Dipl.-Inf. Marcus Hähnel

To my wife, Tina, and my parents, Sylke and Ulrich.

Topic

Intel's RAPL counters are a CPU-provided energy-monitoring mechanism that enables the measurement of CPU energy usage of various CPU components. In previous work it has been shown that it is feasible to use this mechanism to account energy usage of individual tasks in the setting of a single-core system running a multi-tasking OS. The goal of this thesis is to find, develop, and implement a mechanism to achieve the same on a multi-core system.

A challenge of providing accounting for isolated tasks on multi-core systems is, that the RAPL mechanism works on a per socket basis and as such cannot distinguish between tasks running at the same time on the same socket. The thesis should investigate suitable methods to circumvent this limitation and implement necessary changes to the Linux scheduler to allow reasonable precise energy accounting of multi-threaded programs on multi-core CPUs where several threads run concurrently and are not serialized.

The developed mechanism should be evaluated regarding overhead and measurement precision on a recent Intel system using state-of-the-art multi-core benchmark suites.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 16. März 2016

Declaration

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

Dresden, March 16, 2016

Till Smejkal

Abstract

With the evolution of computers towards portable devices, energy consumption became a major design criterion for the development of modern hardware and software. Consequently, being able to determine the energy consumption of the system is an important functionality. While many different approaches to measure the energy of the whole system or even separate components of it already exist, gathering energy statistics for individual programs or groups of processes remains a difficult task. Especially in conjunction with current multi-core devices, which provide a high parallelism to the system, program-accurate energy measurements are very problematic.

In this work, I present a new approach which allows accurate and low-overhead energy measurements of individual programs or groups of them on commercial off-the-shelf multi-core hardware. The measurements are based on the energy reported by the RAPL energy counters available in recent Intel® processors and utilize a special scheduling algorithm to make accurate accounting possible. During the evaluation I show, that this approach can be used to determine the energy of multiple benchmarks from the *NAS Parallel Benchmark* suite with less than 5 % error and less than 6 % runtime overhead in different situations. This high measurement accuracy and low runtime overhead make this energy measurement technique superior to other already available methods.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Within the last decades, computers developed from slow, room-filling machines to small, multi-Giga-Hertz, multi-core devices carried around all the time. In conjunction with this development, the energy consumption of computers became more and more important for the software as well as the hardware design process. Being able to reduce the energy consumption of a system can significantly increase its usability and thereby its market share. Consequently, many different energy measuring techniques were developed during this time period, ranging from direct measurements [10, 22, 13, 23] to model-based estimations [21, 36, 33, 32]. Recently, processor manufacturers such as Intel® and AMD® introduced energy measurement facilities into their processors so that live energy measuring and limiting is possible as well [18, 5]. Still, being able to determine the energy consumption of individual programs running in the system remains a difficult problem. Especially modern multi-core devices, which provide a high parallelism to the system, are very problematic for detailed and accurate energy measurements.

Within the last years, various approaches have been presented by researchers which try to resolve this problem. Some of them, such as *Powerscope* [7] or *eprof* [27] use energy and program traces to calculate the energy consumption of individual programs. Other approaches like LEA^{2}P [29] use special energy measurement hardware in combination with program tracing to achieve a similar result. However, all of these approaches can either not be used to live measure the energy consumption of individual programs or need specially designed hardware to accomplish this.

In this thesis, I present a new, accurate, and low-overhead approach to measure the energy consumption of individual programs in a commercial off-the-shelf multi-core system. As basis for these measurements, I utilize the energy estimations reported by the RAPL energy counters available with recent Intel® processors [18]. Previous research could already show that these energy counters can be used to measure the energy consumption of the system with a high accuracy of up to 97 percent [10, 11]. Unfortunately, the counters only report energy for the whole processor socket in total. Hence, using the counters in a multi-core system, where multiple programs can run in parallel on different processor cores, is problematic. To be able to utilize the RAPL counters for per-program energy measurements, I developed and implemented a special scheduling algorithm, which ensures that only one program with all its threads can use the whole processor at a time. This property enables me to account the whole energy consumed by the processor, which is reported by the RAPL energy counters, to the one program allowed to execute. While in this thesis I only present an implementation, which can measure energy for individual programs, this approach can also be used to gather energy statistics for groups of processes or even whole virtual machines.

During the evaluation, I use the presented measurement technique to determine the energy consumption of various benchmarks from the *NAS Parallel Benchmark* suite [24]. The benchmarks are measured when executed in three different ways: **a)** solely in the system, **b)** together

with other programs in the background, and **c)** in parallel to a second, also measured benchmark. The results of these measurements show, that the presented approach introduces a measurement overhead of less than six percent while never having an energy measurement error of more than five percent. Such a measurement accuracy and low overhead is superior to other already available methods.

The remaining thesis is structured as follows: In the next chapter, I introduce all required technical background so that it is possible to follow the argumentation of the remaining parts of my thesis. Furthermore, this chapter also contains a comparison of my work with other related research. In Chapter 3, I introduce the general idea of the *Locked-Out Gang Scheduling* algorithm as well as explain the energy measurement technique. In addition, I outline in this chapter how these techniques can be integrated into the infrastructure of the Linux scheduler. Chapter 4 contains various details about the implementation of the different techniques from Chapter 3 in the Linux kernel. Within Chapter 5, I present an extensive evaluation of my work with various different experiments. Finally in Chapter 6, I conclude my thesis and additionally outline possible improvements which can be realized in future work.

2 Technical Background

For the ability to accurately account energy in a multi-core system two main capabilities are required. First of all, it is necessary to gather as accurate as possible energy measurements of the system or even preferably of its individual components. Based on these measurements, the next step is then, to apportion the measured energy to individual entities. The entities can be different objects in the system such as users, processes, threads, or even arbitrary groups of them.

In the following I will to give a short insight into the different techniques, which are used currently or which have been used in the past, to achieve any of the two steps just mentioned.

2.1 Energy Measurement Techniques

Measuring the energy or power consumption of modern hardware is a challenging task. Fortunately, researchers have investigated in this area for more than twenty years and came up with many different techniques ranging from direct on-the-chip measurements to model-based estimations. The next subsections will give an overview on the different techniques and will show their advantages as well as disadvantages. At the end I will introduce the measurement technique, which I used during my thesis and explain why this technique was most suited for the problem, which I wanted to solve.

2.1.1 Physical Measurements

A technique, which is often used to determine the energy or power consumption of a computer, is to make physical measurements directly at the machine. For this purpose, different measurement approaches are available.

One possibility to gather energy statistics for a device is to use a multimeter or a comparable tool to measure the power drawn at the power supply unit, at the power distribution unit, or directly at the wall outlet of the computer [10, 22, 13]. Such *AC-measurements* deliver very accurate results with a low error and can be recorded at a frequency of up to 20 samples per second. Unfortunately, while these measurements are suitable to determine the total energy consumption of the whole system, they are not applicable to identify the energy consumption of individual components in the system.

Alternatively to AC-measurements, one could use a different method to measure the power consumption of a system. This approach uses for example *shunt resistors* [21, 20, 23, 3] or *hall-effect based current transistors* [10, 22] in combination with a multimeter or a data acquisition tool to determine the current of the supply lines of the individual components in the system. As the voltage required by these components is normally fixed and predefined, it is possible to calculate the power consumption of the components based on their measured current

drawn. These so called *DC-measurements* allow very detailed and accurate power and energy statistics of a system. With an appropriate system using this technique, it is possible to get power consumption statistics with a sampling frequency of up to 100 samples per second of all the individually measurable components of the device [10].

Unfortunately, physical measurements such as the mentioned AC- and DC-measurements have two significant drawbacks. Firstly, for their evaluation it is necessary to have an additional monitoring entity, which collects and correlates all the measurement points. Most of the time another computer in addition to the measured device is used to perform the analysis of the sampling points gathered by the individual techniques. Consequently, physical measurements can only be used reasonably in scenarios where this additional monitoring entity is acceptable. Secondly, physical measurements, especially DC-measurements, require the ability to insert the measurement probes into the device. While this requirement is already problematic for normal desktop computers or servers, it is even more difficult if one wants to instrument a modern mobile or embedded device.

To sum it up, physical measurements offer very accurate and detailed energy measurements but have requirements on the device under test as well as the measurement environment, which are difficult to fulfill, which is why I consider them inapplicable for the aim of this thesis.

2.1.2 Model-Based Estimations

Instead of directly measuring the energy or power consumption at the device under test, as described in the last section, the idea behind *model-based estimations* is to determine the energy consumption of the device from other measurable factors. One common approach is to collect different events in the system and then calculate with the help of a previously developed and trained energy model the actually consumed energy.

The models as well as the events, which are used in the model-based approaches, are widespread. Some researchers based their models on the performance counters provided by modern processors [21, 36, 33, 32, 31]. The advantage of using performance counters as their model basis is that support for collecting them is well established in current operating systems like Linux. However, as performance counters are only available for the processor and some other selected components of the computer such as the memory controller, estimating the energy consumption of other parts of the device under measurement such as the hard drives or the network card is not possible.

To circumvent this problem, Pathak et al. [27] rested their energy model not on performance counters but on which system calls are used in the system. Because nearly every interaction with the hardware is driven by a system call, this approach also allows a proper modeling of the power consumption of other components than only the processor. A similar approach has also been taken by Shye et al. [30] and Zhang et al. [37]. They based their models on usage information of the components, which are provided by the operating system. Thereby, these models are also capable of estimating the energy consumption of many different parts of the device under test such as the network adapter or the display. The only limitation is whether the operating system provides any information about the component, which can be correlated to its energy.

Unfortunately, model-based estimations have a significant disadvantage. The problem is, that the applied energy model for the specific device has to be trained before one can use

it to estimate the device's energy consumption. This training step of the model can be very extensive as for example with the model described by Snowdon et al. [33]. In addition, this training process most of the time also requires physical measurements at the device, though normally coarse grained AC-measurements[1] are sufficient for this purpose.

In total, model-based estimations provide a promising possibility to collect energy information about a device. Measurements can be done at the level of individual components and only have a small measurement error of about five percent compared to physical ones [27]. Additionally, it is not necessary to have an external monitoring entity as required with physical measurements, but the monitoring process can be integrated directly in the device under test. However, the necessity to train the energy model before it can be used disqualifies the model-based approach for the aim of this thesis.

2.1.3 Simulation-Based Estimations

The idea behind simulation-based estimations is similar to the one of model-based estimations presented in the previous section. The energy consumption of the device is estimated with the help of an energy model and some special events, which can be collected with a small effort. However, instead of using performance counters or system call traces, simulation-based estimations utilize the information provided by a simulator for the device under test [9, 1].

The usage of events from a simulator as basis for the energy model allows very fine grained and detailed energy consumption statistics. Tan et al. [34] for example were able to report energy statistics for individual assembly instructions with their system.

Although this technique may be interesting for debugging purposes, simulation-based estimations are not feasible for the aim of this thesis as I want to do live energy measurements for commercial of-the-shelf hardware. Additionally, this approach also has some essential drawbacks. First of all, the energy model requires, similar to the model-based approach, a training phase before it can be used to estimate any energy consumption. For this purpose, either physical measurements at comparable hardware are necessary, or the energy model must be derived from energy and power information provided by the manufacturer of the device under test. Secondly, it is very difficult to also estimate the energy consumption of other components than the processor and the memory subsystem with a normal simulation-based approach. To be able to gather energy statistics about other components additional energy models for them are required.

2.1.4 On-Chip Energy Counters

With the increasing demand for energy awareness in the embedded and mobile computer sector, processor vendors such as Intel® and AMD® added energy and power measurement functionalities to their processors, which can be used to limit the devices power consumption. A byproduct of this power limiting functionality is, that one can query the internal power and energy measurements from the processor and thereby use them for other purposes as well. The extension which provides this functionality for Intel® processors since the "Sandy Bridge" family is called *running average power limit* (RAPL) [18]. AMD®'s processor extension

[1] See Section 2.1.1 for the different AC-measurement techniques.

which allows monitoring the power consumption of the computer is called *application power management* (APM) and is available since their processor family 15h [5]. In the following I will only concentrate on Intel®'s RAPL and not on AMD®'s APM.

The main purpose of RAPL is to give the operating system the possibility to limit the energy consumption of the processor either for thermal management, power limiting, or power/performance budgeting. To use this functionality, the operating system must define an energy budget as well as a time window, during which this budget should not be exceeded. The processor will then throttle itself as soon as it passes over the defined budget. Fortunately, a side-product of this energy limiting facility is that the processor internally calculates its energy consumption and reports these calculations also to the operating system. Hence, the operating system can to use these energy values to collect online energy statistics for the device it is currently running on.

The energy consumption, which is reported by the processor is no physical measurement, but is estimated based on a set of architectural events combined with power weights [28]. Hence, RAPL calculates the energy consumption via a model-based approach as presented already in Section 2.1.2. However, because the model is directly integrated into the processor, no training of the model is necessary before one can use this technique.

Since its release, various researchers have already worked with RAPL to perform live energy measurements on general purpose computers [12, 2]. In their work Dongarra et al. [6] could show that, although RAPL is only a model-based estimation of the energy consumption of the processor, its reported values are nearly as accurate as the results of physical measurements. This result was further confirmed by Hackenberg et al. [10, 11], which could show that energy measurements done with the latest version of the RAPL energy counters deviates by less than 3 W from simultaneously taken AC-measurements.

All in all, the combination of high accuracy, simple usage, no additional requirements, and its wide availability in modern hardware makes RAPL a very interesting candidate to perform hardware-based energy accounting for multi-core systems as I want to do within this thesis. Unfortunately, RAPL also has some limitations. One of them is, that the energy consumption, which is reported by the energy counters, is restricted to the processor, the internal graphics unit, and the memory controller. Another limitation is, that the energy counters are only updated every millisecond which is a long period of time for recent processors and systems. Still, I consider RAPL the best approach of the four presented techniques for the aim of this thesis.

2.1.4.1 Using RAPL

With the latest version of Intel®processors, RAPL offers four distinct energy counters to the operating system[18], which can be found in Table 2.1.

All four energy counters are updated approximately every millisecond by the processor. Hähnel et al. [12] could show in their work that this update rate is, besides some outliers, which they account to the *system managment mode*, very stable. Unfortunately, it is not possible as a user of the energy counters to determine when the counters were last updated by the processor. This property results in the problem, that the read energy counter values must always be considered outdated. The only possibility to get the very latest energy consumption value is to constantly get the current value from the counter until it changes as shown in

Table 2.1: Description of the different RAPL energy counters.

Counter	Description
PP0	The PP0 energy counter reports the energy consumption of the processor core. This includes all the processing units, caches, and pipelines within the processor.
PP1	The PP1 energy counter reports the energy consumption of a specific device in the uncore, which is usually the integrated graphics unit of the processor. This unit is required for hardware accelerated graphics and only available in the client versions of the processor. Server versions of the processor do not report any value at this counter.
DRAM	The DRAM energy counter reports the energy consumption of the memory controller and all directly attached DRAM units [18]. Previously, this counter was only available in the server version of the processor. However, since the Haswell processor generation Intel® added support for this counter also to the client versions of the processors.
PKG	The PKG energy counter reports the energy consumption of the whole processor socket. This contains the consumption of the other counters as well as additional uncore components.

Listing 2.1. However, when using this approach one must keep in mind, that reading the energy counter also costs energy and thereby influences the final value.

```
unsigned int read_exact(struct rapl_counter c) {
    unsigned int dummy, value;
    dummy = value = read(c);

    while (dummy == value) {
        value = read(c);
    }

    return value;
}
```

Listing 2.1: How to read exact RAPL energy counter values.

The energy counters themselves report the accumulated energy consumption since their last reset. Because the counters are only 32-bit values, they overflow after some time which must be handled during their usage. To get the actual energy consumption reported by the energy counters in a reasonable unit like joules, one has to multiply the read value with an energy domain specific unit as shown in Equation 2.1.

The counter unit as well as the energy counters themselves are presented to the operating system via *model specific registers* (MSRs) of the processor. Hence, reading the latest energy counter value can be done as shown in Listing 2.2. Since reading MSRs is a privileged instruction, the presented code snippet can only be used when the processor currently runs in the privilege level 0, which is usually only the case when the operating system kernel executes. To be able to read energy counter values as a common user program, which are normally

$$E = v_{\text{ctr}} * \frac{1}{2}^{v_{\text{unit}}} \tag{2.1}$$

E	The accumulated energy reported by the energy counter in joules.
v_{ctr}	The value of the MSR for the energy counter.
v_{unit}	The value of the MSR for the unit.

executed with the processor running in privilege level 3, support from the operating system is required. The Linux kernel for example allows reading MSRs as root user with the help of a special file in the *devtmpfs* file system. This file gives access to all MSRs of a processor. Simply by seeking in the file to the corresponding position of the wanted MSR and reading a 64-Bit value, a normal user program can gather the latest energy counter values as well.

```
unsigned int read(struct rapl_counter c) {
    unsigned int value, dummy;

    /* Only the lower 32 bits of the register are of interest. */
    asm volatile (
        "rdmsr"
        : "=a"(value), "=d"(dummy)
        : "=c"(c.msr_nr)
    );

    return value;
}
```

Listing 2.2: How to read the current value of RAPL energy counters.

2.2 Energy Accounting Techniques

While pure energy measurements are already interesting during the development or usage of a device, in modern systems it is even more interesting how much energy a user, a program, or even a function consumed. Such detailed statistics would allow users as well as developers to improve the energy consumption of their tools and thereby create a better user experience in addition to reducing the overall energy footprint. To achieve such a functionality, it is necessary to charge the measured energy consumption to the individual users, programs, or functions. For this purpose again different techniques exist, which I want to discuss in the following.

2.2.1 Trace-Based Accounting

A very common approach to account measured energy to programs or even functions requires two traces of the system. The first trace contains information about the current energy consumption. The second one holds information about which program or function was running at which point in time. During an offline analysis one can then combine the two traces and

thereby generate detailed statistics about which program or function consumed how much energy.

Such an approach is used in the work of Ge et al. [8], Pathak et al. [26], Flinn et al. [7], Chen et al. [4], and Joseph et al. [20]. Unfortunately, trace-based accounting does not allow generating real-time statistics, but only enables analyses after the collection process is finished. Consequently, I consider this approach not acceptable for the aim of this thesis, as being able to gather real-time energy statistics about programs is a very valuable characteristic for a hardware-based energy accounting tool.

2.2.2 Activation-Based Accounting

Alternatively to using traces and offline analyses to calculate energy statistics for programs or individual functions, one could apportion the measured energy consumption directly during the execution to the currently running program or function on the device under measurement. This online apportion can for example be achieved with the help of the operating system scheduler, a process monitoring tool, or a dynamically loaded energy attribution library. Because the energy statistics are created and collected while the monitored program or function executes, the activation-based approach can already provide information about the consumed energy during the runtime of the program or function. This property enables the user or operating system to live monitor the energy behavior of their processes, which is a very interesting functionality for mobile devices like laptops or smartphones.

LEA^{2}P, the work of Sebastian Ryffel [29], for example uses such an approach to account physically measured energy to individual programs on a Linux system. Weaver et al. [35] also use activation-based accounting in their work to attribute energy to individual processes running in the system.

Unfortunately, activation-based accounting also has a drawback. As the attribution of energy to entities is done during the runtime of the device, the consumed energy as well as the available compute performance is influenced by it. Hence, every measurement done with this technique will be erroneous. However, Ryffel could show with measurements that the overall influence on the system can be kept below five percent [29].

In total, I consider the activation-based approach most suited for the aim of my thesis, especially as it supports live energy statistics and is well combinable with on-chip energy counters like the one available with Intel®'s RAPL feature.

2.2.3 Symbolic Execution

A completely different approach is taken by Hönig et al. with their projects SEEP [14, 15] and PEEK [16]. They propose to use symbolic execution of a program in combination with an appropriate energy model or detailed energy measurements on a special measurement device to determine the energy consumption of the program. This combination allows them to find energy hot spots at a function level granularity in an application already during the development process. Hence, their work can be nicely integrated into the development environment and support the programmer with useful energy statistics about his program, which makes writing energy-aware applications significantly easier.

Unfortunately, although symbolic execution can provide very detailed energy consumption statistics, it is not applicable for my work, since I want to measure the energy consumption of existing and actually running programs in a commercial of-the-shelf multi-core system.

2.3 Related Work

The field of research to which my thesis belongs provides many different techniques dealing with energy measuring, energy accounting, or energy management. In the following I will present some publications, which most closely related to the aim of my thesis and compare them with my work.

2.3.1 LEA^{2}P

"The Linux Energy Attribution and Accounting Platform" was developed by Sebastian Ryffel during his diploma thesis [29]. It describes a technique, which combines physical measurements on the device under measurement with an activation-based accounting to programs. For this purpose, he designed a physical measurement system, which is directly evaluated by the device under test as well as an accounting technique based on information provided by the Linux scheduler. The combination of both allows fine grained online energy statistics with a low overhead in a single-core computer. In the case of a multi-core system, he has to split the measured energy onto the different CPU cores, for which he uses a performance counter based energy model.

In comparison, my work utilizes the RAPL energy counters and an extension to the Linux scheduler to generate fine grained energy statistics for individual processes in the system. The usage of the RAPL counters enables my approach to be used on every computer with these counters and not only on the one specially instrumented device. In addition, I do not split the energy for the whole processor into the individual processor core energy consumptions to support multi-core systems, but instead only schedule one process at the whole CPU at a particular time. The proportional accounting technique used by LEA^{2}P to support multi-core systems requires a detailed and trained energy model, which has to be adapted every time the underlying hardware changes. My approach on the other side does not require any adjustments if the underlying hardware changes but can be used in the same way on any hardware having RAPL energy counters.

2.3.2 E-PAPI

Weaver et al. [35] proposed in their work an extension to the "Performance API", which is widely used in the high performance computing area to gather performance statistics of a program. Their proposed extension allows the integration of energy statistics into the existing statistics collection process. This technique would enable a program already instrumented with the PAPI-library to simultaneously collect energy statistics. As sources for the energy statistics everything can be used, which can deliver an energy consumption of the computer, like physical measurements, energy counters such as the RAPL or the APM counters, as well as

other model-based energy estimations. Furthermore, E-PAPI also supports combining energy statistics from multiple sources.

Compared to my work, Weaver et al. provide more like a generic collection framework than an energy measuring tool, which is what I implemented. In addition, their work is not by design able to properly account energy in a multi-core system and would thereby report incorrect energy statistics if multiple programs run in parallel. My approach, on the other side, tries to solve exactly this problem of accurate energy accounting to individual programs in multi-core systems.

2.3.3 Multi-RAPL

Multi-RAPL is a work of myself, during which I used the RAPL energy counters to generate energy statistics for individual processes in a Linux system. However, the approach, which I used to account the measured energy to the processes, is not applicable in a multi-core system but only works on a single-core one. The problem is that Multi-RAPL runs separately on each processor core and hence incorrectly accounts energy if multiple programs run in parallel. Consequently, the work presented in this thesis can be seen as an extension of the Multi-RAPL approach to multi-core systems.

3 Design

As described already earlier, the task, which I want to solve within this thesis, is to realize accurate hardware-based accounting of energy to individual programs in a multi-core system. While the energy measurements should be taken with the help of Intel®'s RAPL energy counters, the accounting is supposed to be based on the Linux scheduler. Within this chapter, I will discuss how it is possible to use the regularly updated RAPL energy counters in a time-multiplexed environment and what changes are necessary in the Linux scheduler to be able to account the measured energy consumption to individual programs in a multi-core system.

3.1 RAPL Counter Multiplexing

In the previous chapter I already explained shortly how the RAPL energy counters work in general and how their values can be obtained by the operating system [1]. Unfortunately, since the energy counters are only a byproduct of the energy management functionality available with the RAPL feature, some difficulties arise if one wants to use these counters to measure the energy consumption of individual entities like programs in a time-multiplexed system. In such a system resources like the processor or memory are constantly shared between all available programs. But, since only one program can use the resource at one specific point in time, the system multiplexes the resources between all programs. Hence, if there are multiple programs, which want to use the same resource simultaneously, access to the resource is granted alternately to the different programs. A result of this resource sharing is, that if there are multiple programs, which want to execute code on the processor, the system will time-multiplex the processor between them. Unfortunately, as the RAPL feature is primarily used by the processor to manage its energy and power consumption and was not designed with the program measurement use case in mind, the energy consumption of all programs, which currently share the processor, is reported by the RAPL energy counters. Accordingly, to be able to utilize the counters for energy measurements of individual programs, it is necessary to multiplex them between the programs as well. In the following I will discuss different resource sharing scenarios and the approaches, which can be used in conjunction with them.

3.1.1 Single Process System

In a simplified environment where only one program can run at one particular point in time and no time-multiplexing happens at all, measuring the energy consumption is very simple. As indicated in Figure 3.1, the consumed energy of any program can be calculated as the difference between an energy measurement taken before the process starts and another one

[1] See Section 2.1.4 for a detailed introduction of the RAPL energy counters.

taken after the process finishes. If there are no other programs allowed to execute in parallel, the consumed energy belongs completely to the currently running program.

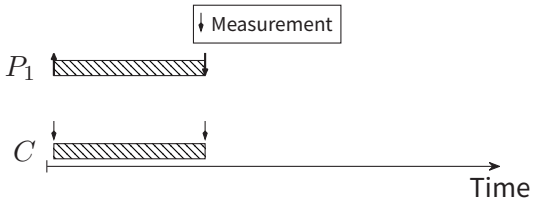


Figure 3.1: Measure the energy consumption of the solely running program P_1 executing on the processor core C.

Since the RAPL energy counters can also be accessed through a special file in the *devtmpfs* file system by a normal user program in a modern Linux system, no additional support in the scheduler or the kernel is necessary to perform energy measurements in such a system. A measurement tool like the one outlined in Listing 3.1 can already provide useful energy statistics for arbitrary programs.

```
energy measure_energy(char* program, char* args[]) {
    rapl_counter_values v_before, v_after;

    read_rapl_counter_values(v_before);

    pid_t child = fork();
    if (child == 0) {
        /* Child */
        execve(pogram, args, NULL);
    } else if (child > 0) {
        /* Parent */
        waitpid(child);

        read_rapl_counter_values(v_after);

        return calc_energy(v_before, v_after);
    } else {
        /* Error */
        throw exception("Error while forking.");
    }
}
```

Listing 3.1: Measure the energy consumption of a solely running program.

Unfortunately, the approach still has one limitation, which comes into existence, if the measured program has an execution time close or below the 1 ms update rate of the RAPL energy counters. In such a scenario more care has to be taken during the reading of the counter values. As shown in Figure 3.2, it is necessary to wait for a energy counter update before the program can be started so that the beginning of the program is aligned with an update. Otherwise, energy consumed by other programs, which executed before the currently measured one, would be part of the final energy calculation. After the program terminates, the next energy counter update has to be waited-for as well so that the termination of the program is also aligned with an update. The actual energy consumption of the program can

then be calculated according to Equation 3.1.

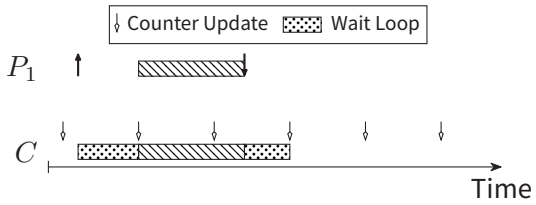


Figure 3.2: Measure the energy consumption of the short-running program P_1 executing on the processor core C.

$$E_{\text{prog}} = energy(v_{\text{end}} - v_{\text{begin}}) - (t_{\text{loop}} * E_{\text{loop}}) \tag{3.1}$$

E_{prog}	The actual energy consumed by the program.
$energy$	A function transforming energy counter values into energy as described in (2.1).
v_{end}	The energy counter values after the program finished.
v_{begin}	The energy counter values before the program started.
t_{loop}	The time spent waiting for the counter update after the program termination.
E_{loop}	The average energy consumption of the wait loop.

Hähnel et al. [12] could show that this approach works reliable for energy measurements for programs or even single routines as long as the execution time is not shorter than half the update interval. In such a scenario, the error introduced by the waiting for the energy counter update increases significantly and hence renders the measurements useless. To be able to measure such short programs with execution times below 400 µs, a measurement technique with a higher temporal resolution is necessary.

3.1.2 Multi Process System

As soon as multiple programs can run simultaneously in the system, the simple approach as explained in the last section is not applicable anymore. Since the scheduler of the system multiplexes the processor between all available processes, the energy consumption calculated with the simple approach is useless, because the measured energy consumption would be a combination of all programs, which were executed together with the measured one. Figure 3.3 illustrates the corresponding problem with an example where two measured programs share the processor.

The simple approach explained in the last subsection assumes that all the energy consumed between the start and the termination of a program belongs completely to it. Unfortunately, with program P_2 becoming ready, the processor is shared between the two programs by the scheduler. Hence, the energy consumed from the start of P_2 to the termination of P_1 is a combination of the individual energy consumptions of the two programs as well. However, the simple approach does not handle such scenarios correctly and instead accounts, as shown

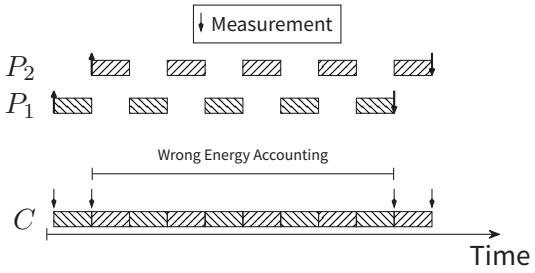


Figure 3.3: Problem with the simple energy measurement approach in a time-multiplexed system if the processor core C is shared between two programs P_1 and P_2.

in the figure, the whole energy consumed during this time period to P_1 and P_2, which is not correct.

Consequently, a different approach must be taken to still be able to gather energy statistics for individual programs. One possibility is to integrate the energy measuring and accounting into the scheduling process. Instead of reading the RAPL energy counters only before the start and after the termination of a program, one must also evaluate the counters at every switch between two programs as shown in Figure 3.4. The energy, which is consumed between two subsequent counter evaluations, must be accounted to the program, which has been executed during this period of time. At the termination of the program, its final energy consumption can be calculated as the sum of all partial measurements taken during its whole execution time.

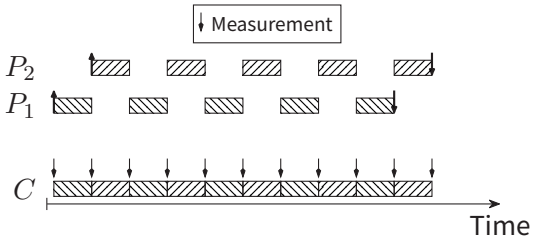


Figure 3.4: Measure the energy consumption of two simultaneously running programs P_1 and P_2 if they share the processor core C.

Multi-RAPL[2] implements exactly this approach to measure the energy consumption of individual programs in a time-multiplexed system. It integrates the energy measuring and accounting in the scheduling process of the Linux kernel and thereby is able to accurately measure energy in a system where the processor is shared between multiple programs. To achieve this functionality, Multi-RAPL saves for every process in the system how much energy it consumed during its execution and updates this information every time the process gets removed from the processor due to a switch in the scheduler. Since it also uses the previously explained approach to determine the energy consumption of programs running shorter than the RAPL counter update rate[3], energy is accounted accurately even if switches between processes happen very often.

[2] Multi-RAPL is explained in more detail in Section 2.3.3.

[3] In Section 3.1.1 I describe how the energy consumption of short-running programs can be measured accurately.

3.1.3 Multi-Core Systems

Until now, I only considered energy measuring and accounting on single-core systems. Unfortunately, since the energy consumption reported by the RAPL counters is not separated for each individual core on the processor, energy accounting on multi-core systems is significantly more difficult than on single-core systems. Like illustrated in Figure 3.5, the Multi-RAPL approach presented in the last section cannot be used in such a system. Multiple programs can run in parallel in the system on different cores without the core-local scheduler noticing. This parallelism of the processor available with multi-core systems renders the measurements taken with the Multi-RAPL approach useless.

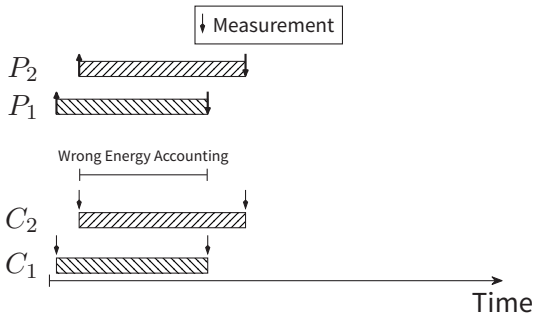
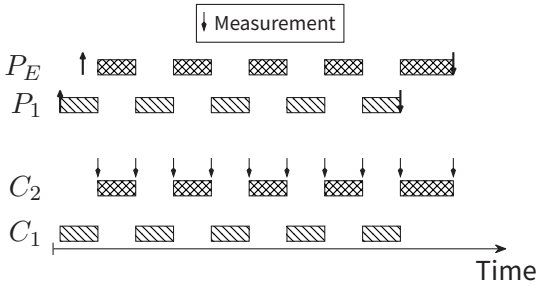


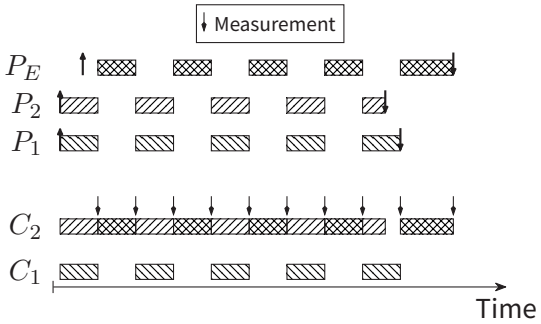
Figure 3.5: Problem with the Multi-RAPL energy measurement approach if the measured programs P_1 and P_2 run in parallel on the two processor cores C_1 and C_2.

Every time when two programs are executed simultaneously on different processor cores, energy accounting solely based on the RAPL energy counters is not possible anymore, since one cannot determine which program consumed which portion of the overall processor energy consumption reported by the energy counters. The approach used by Multi-RAPL does not respect such scenarios but instead assumes that all the processor energy is consumed by the currently running program. Hence, as shown in Figure 3.5, during the time period when both programs P_1 and P_2 run in parallel on the different processor cores C_1 and C_2, the energy is not correctly accounted. Multi-RAPL does not split the total energy between the two programs, but charges the whole energy to P_1 and P_2.

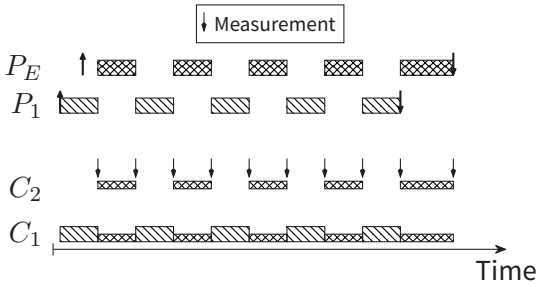
Accordingly, in a multi-core system a different approach must be used to achieve program-accurate energy measurements. The technique, which I developed within this thesis, makes correct measurement possible in such a system by restricting the usage of the processor to only one measured program at a time. Hence, parallel execution of two individually measured programs or of a measured program and another not measured one on different cores of the processor is not allowed. This restricted parallelism gives me the possibility to properly account the consumed energy to the individually measured programs. Unfortunately, this restriction of the parallelism also implies that the available performance of the system is reduced as only one program at a time can use the processor whereas in a normal multi-core system multiple programs could run in parallel. Consequently, I only enforce this restriction if and only if a measured program is currently running on the processor. If there is no program measured at the moment, other programs are still allowed to run in parallel. Additionally, I also allow internal parallelism of the measured program. Which means, that if a program



(a) Measure the single-threaded program P_E while one other program is running.



(b) Measure the single-threaded program P_E while two other programs are running.



(c) Measure the multi-threaded program P_E while one other program is running.

Figure 3.6: Measure the energy consumption of the program P_E executing together with other programs in a multi-core system.

internally uses threads to perform actions in parallel, I allow parallel execution of them on multiple cores as accurate energy measurements for the program itself are still possible.

In Figure 3.6 it is shown how the approach, which I implemented for this thesis, works in general. Every time when the measured program P_E is running on the processor only threads belonging to this program are allowed to run and all the other programs must wait. This technique ensures, that the energy, which is measured between two subsequent switching points, is only consumed by P_E and not some other parallel running program. The threads of P_E can spread across all available processor cores and are allowed to run in parallel because all the energy they consume is accounted to their associated program. However, as soon as the measured program finishes its execution, the other programs are allowed to run again and as it is not necessary to run them solely on the processor, they are allowed to run in parallel as well. In total, this approach ensures that energy can be correctly accounted in a multi-core system while not sacrificing the system's performance and parallelism too much.

3.2 The Linux Scheduler

Since the aim of this thesis is to perform program-accurate energy measurements on a multi-core system, I implemented the RAPL counter multiplexing as described in Section 3.1.3 to achieve this goal. A necessity to reach this goal is to integrate the proposed scheduling scheme into the Linux scheduler environment so that correct energy accounting and measuring is possible. Before I explain in Section 3.4 in detail how I integrated the new scheduling scheme into the Linux scheduler, I first want give some more background information about the Linux scheduler architecture itself in this section.

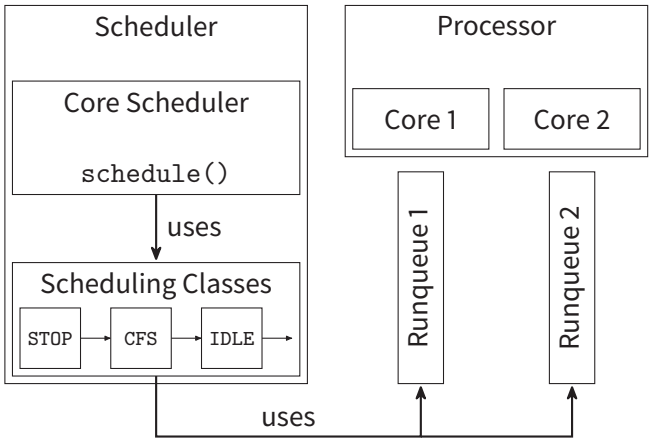


Figure 3.7: Overview of the Linux scheduler architecture.

During the development of the Linux kernel version 2.6.23 in April 2007 [17] the Linux scheduler was completely refactored to its current implementation. Now the scheduler consists of two separated building blocks — the *core scheduler* and the *scheduling classes* — which each perform different tasks in the system. In addition to this vertical separation of the scheduling process, the scheduler is also separated horizontally. For every core of the processor an independent scheduler instance exists. This horizontal separation is represented in the scheduler

by having different *runqueues* for each available core. In Figure 3.7 an overview of all these different parts of the scheduler as well as their interactions is shown.

Core Scheduler The core scheduler implements the main part of the scheduling process in the Linux kernel. This includes for example the switching between execution contexts on the processor, the blocking and waking of processes as well as all the handling of scheduler related system calls.

It also contains the main scheduling routine, which is executed at every scheduler invocation and implements the actual scheduling process — the time-multiplexing of the processor. Within the routine, all available scheduling classes are asked one after another if they want to replace the currently running program on the core. If one class has a program, which it wants to execute instead of the current one, it returns this program to the main scheduling routine, which in turn will result in a context switch on the processor.

Another task, which is implemented in the core scheduler, is the management of process priorities and the assignment of programs to processor cores. To change the priorities of a process the core scheduler either moves the process between scheduling classes or just informs the current class of the process so that it changes its internal management structures accordingly. Movement of processes between processor cores is realized in the core scheduler by removing the process from its current runqueue and inserting it into the runqueue of the new core. This procedure eventually also requires switching between processes if the moved process is currently executing on the processor.

Scheduling Classes The main functionality of the scheduling classes in the Linux scheduler architecture is to manage processes according to their scheduling criteria. In the Linux kernel currently five different scheduling classes exist, which each implement different scheduling approaches. The *Stop Scheduling Class* for example schedules its processes according to a simple FIFO method whereas the *Completely Fair Scheduling Class* uses a more complex mechanism to achieve fair processor sharing between its processes. Internally both of them use appropriate data structures so that the management of their processes is as simple as possible. However, the process management in a scheduling class not only requires the implementation of scheduling decisions according to their internal scheduling criteria, but also the implementation of process addition and deletion, scheduling features like *yield* or *yield to*, as well as accounting of consumed processor time to the corresponding processes.

Within the Linux kernel, scheduling classes were introduced so that the different scheduling priority bands like *real-time*, *idle*, and *normal* can be separated. Each scheduling class handles its corresponding priority band independently of the other classes. Hence, every scheduling class acts as if it is the only scheduling class in the system. It is the core scheduler's task to combine the individual scheduling decisions of each scheduling class. In the Linux kernel this combination of the individual scheduling decisions is achieved by processing them according to the priorities of the classes with the result that the scheduling decision of the highest priority class rules out all other ones.

Runqueues In the Linux scheduler architecture runqueues are the most important management data structure. Each runqueue belongs to one processor core in the system and

contains all the information about all programs, which are currently assigned to this core. Consequently, the runqueue combines all the individual data structures used by the scheduling classes to manage their assigned programs.

This separation of processor cores enables the Linux scheduler to make individual and independent scheduling decisions on each core. In addition, since all required information is stored in the core local runqueues, the same scheduling routines can run in parallel on each core without interfering with each other. That means, that for example the Completely Fair Scheduling Class can make scheduling decisions simultaneously on multiple processor cores.

Furthermore, since runqueues represent the scheduler state on each core, they contain other additional data. For example, the program which is currently running on the processor, the number of processes available on the core, and various usage and runtime statistics. All of them are constantly updated within the main scheduling routine or the scheduling classes to always reflect the current state of the scheduler.

3.3 Locked-Out Gang Scheduling

In Section 3.1.3 I already outlined roughly how a scheduling algorithm should look like to be usable in combination with energy measurements on a multi-core system. The presented scheduling algorithm proposes that while a program is running on the processor, for which energy measurements should be taken, no other program should be allowed to run in parallel. Only the threads, which belong to the measured program, can be executed simultaneously on multiple processor cores. If one wants to implement this scheduling algorithm in the Linux scheduler, various steps have to be performed.

Building Gangs The Linux scheduler internally has no differentiation between programs and threads since the general scheduling procedure does not require this differentiation. Consequently, the Linux scheduler only knows execution contexts which are also called *tasks*. Still, for the purpose of this scheduling algorithm it is feasible to group multiple of these tasks together in a so called *gang* and allow them to execute in parallel so that the performance benefit of a multi-core system can be utilized.

Accordingly, as the aim of this thesis is to measure the energy consumption of individual programs, the implementation of the proposed scheduling algorithm in the Linux kernel must subsume all tasks, which belong to the same user level program in a gang, so that they can run simultaneously. Fortunately, this can be achieved inside the Linux kernel by collecting all tasks, which belong to the same *kernel thread group*. A thread group is the Linux kernel's notation of tasks, which are part of the same user level program. Consequently, the final implementation in the scheduler unites all tasks belonging to one user level program in one scheduling gang. All members of one gang are scheduled simultaneously on the processor as proposed in the scheduling algorithm.

Simultaneous Gang Scheduling As explained in the previous section, the Linux scheduler performs scheduling decisions independently on each processor core. Unfortunately, the proposed scheduling algorithm requires that, if a measured program is going to be scheduled on one processor core, all other cores have to do the same. Hence, it is necessary for the

algorithm that some scheduling decisions are not done independently on each processor core but globally for the whole system.

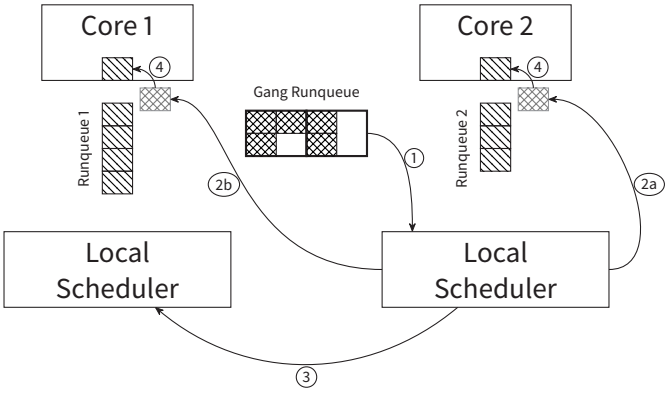


Figure 3.8: Implementing simultaneous gang scheduling in the Linux scheduler environment.

To fulfill this requirement in the Linux kernel, an approach as outlined in Figure 3.8 could be used. The scheduling gangs with all their runnable tasks are managed in an independent runqueue. This separate management of all scheduling tasks belonging to scheduling gangs prevents situations, where tasks are scheduled accidentally not as a whole gang but separately. If on one processor core the local scheduler decides to run a gang, it removes all the gang's tasks from the separate runqueue (1), assigns some of them to its local processor core (2a), and distributes the remaining ones on the other cores (2b). Thereafter, it forces all processor cores in the system into their core-local main scheduling routines (3), which in turn will cause their assigned gang members to be executed instead of the tasks currently running on the core (4). Thereby, all processor cores available in the system will schedule the whole scheduling gang representing one measured program simultaneously as soon as any local scheduler decides to switch to the gang.

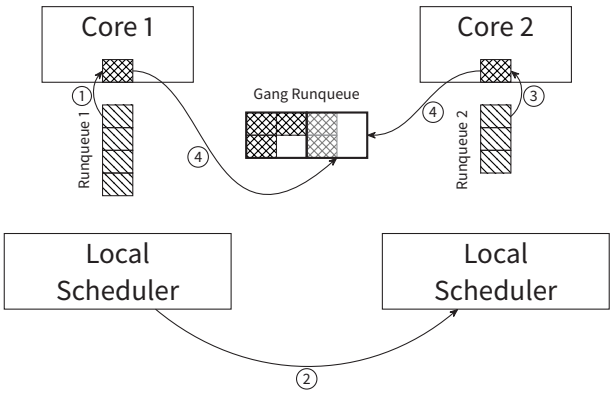


Figure 3.9: Implementing simultaneous gang descheduling in the Linux scheduler environment.

A similar algorithm can also be used to perform simultaneous gang descheduling in the system. As it is shown in Figure 3.9, as soon as the local scheduler of one processor core

decides to stop executing a gang member in favor of a different task, which is not part of the gang (1), it forwards this decision to all other local schedulers in the system (2). They also remove their gang members from the processor core and execute other tasks not belonging to the measured program (3). Finally, all gang members are removed from the normal runqueues and inserted back into the dedicated scheduling gang runqueue (4).

Process Lock-Out In the scheduling algorithm of Section 3.1.3 I propose, that if a measured program executes, either threads belonging to the program run on a processor core or nothing at all. However, on normal hardware it is not possible to run *nothing* on a processor core. This hardware property is also represented in the Linux scheduler. Accordingly, if the gang has not enough members to occupy all cores of the processor, the local scheduler of an unoccupied core will run other tasks, which are not part of the scheduling gang representing the measured program and thereby influence the energy measurements.

Consequently, since I also want to be able to measure programs, which have fewer threads than there are cores in the system, local schedulers, which currently do not have a scheduling gang member assigned to them, must be prevented from executing other non related tasks. The solution, which I use for this problem, is to assign a special task to these unoccupied processor cores. This task does not belong to the currently scheduled gang but acts as if it belongs to it and consequently captures the corresponding core. But instead of executing any user program, the task's purpose is to consume the least possible energy so that the energy measurements for the actually measured program are influenced as small as possible.

3.4 The Energy Scheduler

In Section 3.2 I already explained, that the Linux scheduler internally is separated into the core scheduler and various scheduling classes. Whereas the core scheduler performs all the general scheduling operations such as waking and blocking of programs, execution context switching, and system call handling, the scheduling classes implement the actual scheduling algorithm. Consequently, I decided to introduce a new scheduling class into the Linux scheduler framework — the *Energy Scheduling Class* — which implements the proposed *Locked-Out Gang Scheduling* algorithm as explained in the last section.

3.4.1 The Energy Scheduling Class

To be able to realize the Locked-Out Gang Scheduling algorithm in the Linux scheduler environment, the new scheduling class has to perform various different tasks. In the following I will discuss the most important ones among them.

Process Management One important functionality, which my scheduling class has to provide, is the management of all its assigned processes. A fundamental additional property for the management is, that the Energy Scheduling Class not only operates on scheduling tasks but also on scheduling gangs. Hence, for every operation within the class it is necessary to know, to which scheduling gang each scheduling task belongs.

Consequently, I decided to not only have one management structure for the scheduling class, but two. The first data structure is used to manage the different gangs, which are administrated by the class. This data structure additionally contains information about which scheduling tasks belong to the gang as well as important runtime statistics. Since the scheduling gangs must be managed globally in the system, this data structure is shared between all core-local scheduler instances. The second data structure used by the new scheduling class contains all the information required by the core-local scheduler. Hence, the data structure provides information such as which gang members are currently assigned to the local scheduler's processor core, which scheduling gang is currently executing, as well as where the special idle task, which must be executed if no gang members are assigned to the processor core at at the moment, can be found.

Scheduling Another functionality, which the Energy Scheduling Class has to provide, is to decide when and for how long to run a scheduling task on the processor. In the context of my scheduling class this operation is split into two separate decisions.

First of all, the class must decide when and for how long to run which scheduling gang. For this purpose I use a time slice round robin scheduling scheme. Each scheduling gang gets a time slice assigned, which defines the amount of time the gang's scheduling tasks are allowed to run on the processor cores. The length of the time slice is determined based on the number of gang members. After this time slice is depleted, the next gang in the list of scheduling gangs gets executed. The decision, which gang to run next, also contains another decision, namely on which processor core which gang member should be executed. The approach, which I use in this thesis, is to simply distribute all gang members evenly among the available processor cores. This distribution technique allows proper resource usage in a multi-core system, while still keeping the implementation complexity small for the scheduling class.

The second decision for the scheduling has to be done locally on each processor core. After the global decision for a scheduling gang distributed the gang members to the available cores, the core-local scheduler has to decide, which gang member to run and when. For this purpose, I again use the simple time slice round robin scheduling scheme. Similar to the handling of the scheduling gangs, each gang member gets a fixed time slice assigned, during which it is allowed to execute on the processor. After the time slice is exhausted, the next member assigned to this core can run for its time slice. If there is no other member available on the core, the current one can directly continue with its next time slice.

Energy Measuring and Accounting In addition to the normal tasks of a scheduling class, the Energy Scheduling Class also has to measure and account energy for its managed programs, since this was the reason why I introduced the scheduling class in the first place. As described in Section 3.1.3, to be able to accurately determine the energy consumption of a program, it is necessary to run the program exclusively on the whole processor and to measure only those periods when the program actually executed and not some other program which time-shares the processor with the measured one.

Consequently, for the energy measuring it is necessary to determine when the program executed and calculate the energy consumed during these time periods. This functionality is achieved in the Energy Scheduling Class by reading the RAPL energy counter values every

time a scheduling gang is selected to be executed or removed again from the processor. With the help of these counter values, the energy consumption of the corresponding execution time slice can be calculated. By summing up all these individual partial measurements, the total energy consumption of the program can be determined.

3.4.2 Integration into the Scheduler Environment

As I already explained in previous sections, each scheduling class in the Linux kernel represents its own priority in the scheduling process. Classes, which manage tasks of a higher priority band, are favored over other scheduling classes, which handle tasks with a lower priority, during the scheduling decision procedure. Since I am introducing a new scheduling class into the existing group of classes, I had to decide where to insert this scheduling class into the overall priority hierarchy.

Currently the Linux kernel differentiates five different priority bands in the scheduler. The *Stop Scheduling Class* has the highest priority in the system and manages very important processes executing jobs on behalf of the Linux kernel such as the migration of scheduling tasks between different processor cores. The next two scheduling classes — the *Deadline Scheduling Class* and the *Real Time Scheduling Class* — manage user processes, which have special properties like a limited amount of execution time or a point in time until they have to finish their execution. The *Completely Fair Scheduling Class*, which has a lower priority than the previous two, contains normal user processes. Those normal processes are all kinds of interactive or non-interactive programs, which a user starts on its system. The last scheduling class — the *Idle Scheduling Class* — contains only one scheduling task per processor core which is executed if there is nothing else to run on the core. These idle tasks will transition the corresponding processor cores to a sleep state in order to reduce the overall energy consumption of the system.

Since I want to be able to measure the energy consumption of normal user programs, I decided to insert my new scheduling class between the Completely Fair and the Real Time Scheduling Classes. Thereby, the Energy Scheduling Class can run its tasks in favor of normal user programs, which is a requirement to achieve the process lock-out as well as the simultaneous scheduling of gang members. If I would have added the Energy Scheduling Class with a lower priority than the Completely Fair Scheduling Class (CFS), programs managed by my class could have only be scheduled if no other tasks wants to execute on the processor. This property would have complicated the implementation of the Locked-Out Gang Scheduling algorithm significantly.

Unfortunately, adding the Energy Scheduling Class with a higher priority into the class hierarchy than CFS also has a drawback. One aim of my implementation is, that measured programs can run time-shared with other non measured programs. This property is a necessity to obtain a functional and usable system during measurements. However, since measured programs run with a higher priority than other programs and additionally occupy the whole processor while they execute, normal programs are not able to execute while there are measured processes in the system, which results in an unresponsive system. To prevent this behavior, I added a *scheduling class time-sharing* mechanism in the Energy Scheduling Class. This mechanism ensures that if there are other processes in the system, which do not belong to the Energy Scheduling Class, the my scheduling class gives up the processor from time to

time so that other programs can execute as well. In this way, the system remains responsive although measured programs are executing exclusively on the processor.

4 Implementation

With design as well as general ideas explained in the previous chapter, I would like to point out details of the implementation and integration of my newly introduced Energy Scheduling Class into the Linux scheduler in the following sections. For this purpose, I will first describe how scheduling classes are implemented in general in the Linux kernel as well as what functionality I had to provide so that my scheduling class integrates well into the system. Secondly, I will introduce some important data structures and helper functions, which have been crucial for the realization of the Locked-Out Gang Scheduling algorithm. Thirdly, I will explain the energy measurement and accounting functionality and how the calculated energy consumption of a program is made available to the user. Lastly, I will discuss various problems, which occurred during the development and present how I solved them.

4.1 The Scheduling Class

To be able to perform accurate energy measurements for individual programs on a multi-core system, I decided to introduce a new scheduling class into the Linux kernel, which realizes a scheduling algorithm specially designed for this measuring purpose. Since the scheduling classes in the kernel are only one part of the overall scheduler environment and hence have to interact with various other components of the scheduler, scheduling classes have to be implemented according to a well defined structure. Listing 4.1 gives an overview of this interface of a scheduling class. Because the Linux kernel is written in the programming language *C*, which is not object-oriented, the class interface is implemented as a container of functions. All scheduling classes available in the Linux scheduler are represented as an instance of this container with all the function references pointing to their individual implementations. The containers themselves are combined with the `next` pointer, which allows the main scheduling routine in the core scheduler to iterate through the list of scheduling classes without knowing them directly. If a method in the scheduler wants to use a functionality of a scheduling class, it just has to use the corresponding function reference in the scheduling class' container, which is redirected to the associated implementation in the scheduling class. Hence, algorithms in the core scheduler can be written class independent, as they do not need to know the individual classes, but just the general interface.

In order that a scheduling class can be used correctly within the Linux scheduler, most of the functionality defined in the class interface must be implemented by the class. An overview about the respective tasks and properties of the most important functions of the interface can be found in Table 4.1. Some functions such as the `task_fork`, `task_dead`, and `switched_from` methods are not necessary, but only need to be implemented by a scheduling class, if it wants to perform any internal updates at the corresponding events. Additionally, all the methods, which are inside the `CONFIG_SMP` preprocessor switch, must only be provided

```
struct sched_class {
    const struct sched_class *next;

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);

    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

    struct task_struct * (*pick_next_task) (struct rq *rq, struct task_struct *prev);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
    int  (*select_task_rq)(struct task_struct *p, int cpu, int sd_flag, int flags);
    void (*migrate_task_rq)(struct task_struct *p, int next_cpu);

    void (*task_waking) (struct task_struct *task);
    void (*task_woken) (struct rq *this_rq, struct task_struct *task);

    void (*set_cpus_allowed)(struct task_struct *p, const struct cpumask *newmask);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);
#endif

    void (*set_curr_task) (struct rq *rq);

    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork) (struct task_struct *p);
    void (*task_dead) (struct task_struct *p);
    void (*switched_from) (struct rq *this_rq, struct task_struct *task);
    void (*switched_to) (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio);

    unsigned int (*get_rr_interval) (struct rq *rq, struct task_struct *task);

    void (*update_curr) (struct rq *rq);
};
```

Listing 4.1: The interface of scheduling classes in the Linux scheduler.

by a class if the class should be capable of running in a multi-core system.

Within the Energy Scheduling Class, I implement most of the available functionality provided in the class interface. While some of the defined functions are very simple to implement, others are more complicated. The most important and also most difficult ones among them are described in more detail in the following.

Enqueue and Dequeue Inside these functions I insert or remove a scheduling task from my internal management data structures. These operations always require that the scheduling gang, to which the task belongs, has to be found first, since all changes on a scheduling task must be coordinated with its corresponding gang.

During the insertion, the new task is added into the global gang runqueue[1]. To accomplish this insertion operation, I have to find the task's gang data structure — or if it not yet exists,

[1] See Section 4.2 for more details about the different management structures.

Table 4.1: Description and properties of the most important scheduling class methods.

Method	Description
`enqueue_task`	The purpose of this method is to insert the given scheduling task `p` into the internal management data structures of the scheduling class.
`dequeue_task`	Within this function, the scheduling class has to remove the given task `p` from its internal management data structures.
`pick_next_task`	This routine is called within the core scheduling function and has to return the scheduling task, which should run next or `NULL`, if the class has no task to execute.
`put_prev_task`	If the currently executed task gets replaced by another task on the processor, this method of the current task's scheduling class is called so that the class can adapt its internal information accordingly.
`set_curr_task`	The purpose of this method is to inform the scheduling class that there is a task currently executing on the processor, which belongs to this class, but the decision to run this task on tho processor was not done by the class.
`task_tick`	This function is called on the scheduling class of the currently executing task every time when a scheduling timer tick happens. Within this method the class is supposed to update internal statistics so that it can make appropriate scheduling decisions.
`update_curr`	The `update_curr` function is used if an extraordinary scheduling event happens, during which the internal statistics of the currently running task need to be updated.

create a new one — and append the task to the scheduling gang's list of scheduling tasks. Additionally, I also check whether the corresponding gang is currently executing and correspondingly assign the task to a core-local runqueue as well, so that the task gets executed together with its other gang members.

During the removal of a task, the reverse steps are performed. First, I check whether the task is currently assigned to a core-local runqueue and accordingly remove the task from this data structure. Secondly, the task is also removed from the global runqueue by deleting it from the task's gang data structure. As last step during the removal operation, I test if I have to change the assignment of gang members to processor cores when the gang is currently executing so that load remains equally distributed in the system.

Task Tick Within the `task_tick` method of the scheduling class, I first update internal runtime statistics of the corresponding scheduling task and then decide based upon these statistics, whether I have to reschedule the current task or even the whole scheduling gang. If a rescheduling is required, I force the scheduler into the main scheduling routine by setting a reschedule flag for the local scheduler. Within the main scheduling routine the `pick_next_task` method is called, which then has to perform the actual scheduling decision.

Pick Next Task The `pick_next_task` function in the Energy Scheduling Class contains most of the internal scheduling logic. In the other existing classes like the Completely Fair Class, which just perform core-local scheduling, this method only has to decide between the currently available scheduling tasks and return one of them so that the task gets executed on the processor. However, since my scheduling class has to make system-wide as well as core-local decisions, I have to perform multiple operations within this method.

System-Wide Decision First of all, because the Locked-Out Gang Scheduling algorithm requires that scheduling happens simultaneously on all processor cores, a system-wide decision about what to run next has to be reached. Since I also implement a time-sharing between the Energy Scheduling Class and other lower priority classes, I first decide whether my scheduling class can run and thereby occupy the whole processor or if a different class should be able to schedule its tasks, so that the system remains usable.

If I decide to run my scheduling class, I then have to select which scheduling gang to distribute in the system. For this purpose I use a simple time slice round robin scheduling scheme as already described earlier in Chapter 3.

After I reached a system-wide decision, any changes, like that a new scheduling gang must be executed or that the processor needs to be given up, are passed on to the core-local scheduler instances[2], which then have to perform their local operations. This decision forwarding process also contains the distribution of all gang members of the scheduling gang, which was selected to execute now, to the available processor cores.

Core-Local Decision With the system-wide decision propagated to the local scheduler instances, the second part of the schedule operation starts. If the global decision was to stop executing the Energy Scheduling Class, the local schedulers have to give up the processor by simply returning `NULL` to the main scheduling routine, so that other scheduling classes can take over, Otherwise, if the decision was to execute a new scheduling gang, the local scheduler has to select which one of the assigned gang members should run on the processor. For this purpose, the scheduler again uses the time slice round robin scheduling scheme. In the case, that no gang member was assigned to the current processor core, the special *idle task*, which prevents the execution of other unrelated tasks, is selected to run on the processor core.

Additionally, core-local scheduling decisions can also occur if there was no previous global scheduling decision. Such locally-only scheduling happens, if there is more than one gang member assigned to the corresponding processor core and if the `task_tick` routine recognized that the time slice of the currently executing gang member is depleted. In such a scenario, the `task_tick` method forces the local scheduler into the main scheduling routine, which in turn causes the `pick_next_task` function to execute, which is then able to switch to a different gang member on the current processor core.

Set Current Task This function has a special role within the Energy Scheduling Class. As described in Table 4.1, this method informs the scheduling class that there is a task currently running on the processor, which belongs to the class, but which was not selected by the class

[2] See Section 4.4 for more details, how the decision propagation to the core-local scheduler instances is achieved.

to be executed. Unfortunately, the Locked-Out Gang Scheduling algorithm requires that any task, which is managed by the Energy Scheduling Class, has to be executed together with its gang members. However, if the decision to run this task was not made by the Energy Scheduling Class but somewhere else, this crucial property may not be fulfilled.

Consequently, this function has to resolve this inconsistency. For this purpose, the function forces all other core-local schedulers to also execute members of the corresponding gang by globally switching to the task's scheduling gang.

4.2 The Task Management

Besides deciding which task to run next on the processor, the second most important functionality, which a scheduling class has to implement, is the internal management of its assigned processes. Because of the special requirements of my intended scheduling algorithm, I need multiple data structures to implement this management. I require one data structure to administrate the global list of scheduling gangs and their members and another data structure for each core-local scheduler instance, where all the assigned gang members of the currently executing gang are organized.

4.2.1 Global Scheduling Gang List

The first data structure which my scheduling class requires to manage its tasks is used to represent a scheduling gang in the system. As shown in Listing 4.2, this data structure contains a list of all its gang members (`runnable`), a reference to the gang leader (`task`), and some additional management information.

```
1  struct energy_task {
2      int state;                          /* Runtime statistics */
3      ktime_t start_exec;
4      u64 exec_time;
5
6      struct task_struct *task;           /* The gang leader */
7
8      struct list_head runnable;          /* List of gang members */
9      u32 nr_runnable;
10
11     struct list_head rq;                /* Link into the global gang list */
12 };
```

Listing 4.2: The data structure used in the Energy Scheduling Class to represent a scheduling gang.

The list of gang members is organized as a simple circular double-linked list of `task_struct` structures, the Linux kernel's representation of an executable entity. If a new task is added to the Energy Scheduling Class, this task gets inserted into the gang member list of the corresponding scheduling gang structure. Since the members among each other are not handled differently inside the scheduling class, neither further information nor a more complicated organization structure is necessary to manage all members of one gang.

The `task` variable in the scheduling gang data structure is used to identify the user process, which is represented by this gang. This information is needed for building up the gang because only with this information one can find all threads belonging to the same user level program. Additionally, during the energy measuring and accounting this reference to the user program is also utilized, since all the consumed energy is accounted to the gang leader of the currently executing gang[3].

The management information such as the `state`, the `start_exec`, and the `exec_time` variables are used for the scheduling decision process. Based on them the usage of the current time slice of the scheduling gang is calculated, which in turn is used to decide which scheduling gang to run.

The individual scheduling gang objects, which each represent a measured program in the system, are combined similarly to the gang members in a circular double-linked list which originates in the data structure shown in Listing 4.3. This data structure represents the global scheduling gang list and thereby is the main management structure of the Energy Scheduling Class.

```
1  struct global_rq {
2      raw_spinlock_t lock;
3
4      int state;                          /* Runtime statistics */
5      ktime_t start_running;
6      ktime_t stop_running;
7
8      struct list_head tasks;             /* List of scheduling gangs */
9      u32 nr_tasks;
10     u32 nr_threads;
11 };
```

Listing 4.3: The data structure of the global list of all scheduling gangs managed by the Energy Scheduling Class.

This structure only contains the list of scheduling gangs (`tasks`) as well as some additional management information. The runtime statistics such as the `start_running` and the `stop_running` variables are used in combination with the statistics about the number of managed scheduling gangs for the time-sharing between the Energy Scheduling Class and the other scheduling classes[4]. Because this global list is shared between all scheduling instances in the kernel, it is necessary to lock-protect the data structure against concurrent changes to it. For this purpose, I use the internal lock variable of the data structure inside the functions of the Energy Scheduling Class to prevent corruption of the scheduling gang list due to not synchronized parallel access.

An example of the overall interaction and combination of the different global management data structures of the Energy Scheduling Class is shown in Figure 4.1. Within the example, a situation is visualized where the scheduling class manages three different scheduling gangs. The first one has three members, whereas the second one has only one member, and the third one has two members. Additionally, the first scheduling gang is currently executed on the

[3] See Section 4.3 for more details about how energy measuring and accounting is realized in the Energy Scheduling Class.

[4] The scheduling class time-sharing algorithm is explained in detail in Section 4.5.

processor, which is why the global scheduling gang list structure as well as the structure of the corresponding gang have a `1` assigned to their `state` variables.

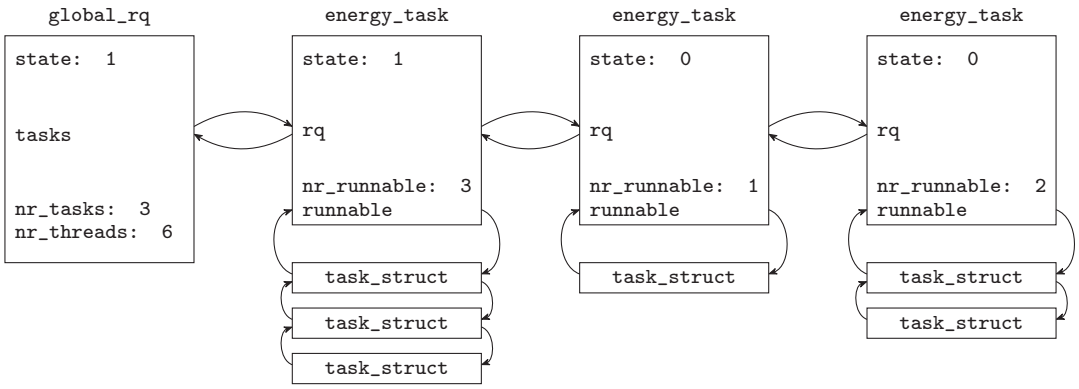


Figure 4.1: Interaction of the global management data structures of the Energy Scheduling Class.

4.2.2 Core-Local Runqueue

```
1  struct e_rq {
2      raw_spinlock_t lock;
3
4      int state;                          /* Runtime statistics */
5      int resched_flags;
6
7      struct list_head runnable;          /* List of assigned gang members */
8      int nr_runnable;
9
10     int nr_assigned;
11
12     struct task_struct *curr;           /* The currently executing gang member */
13     struct energy_task *curr_e_task;    /* The currently executing gang */
14
15     struct task_struct *idle;           /* The idle task */
16 };
```

Listing 4.4: The data structure used by the Energy Scheduling Class to manage all the information required by the core-local scheduler.

For the management of all the gang members, which are assigned to the current processor core, the data structure as shown in Listing 4.4 is used. Since this data structure is also utilized for the core-local scheduling decision, it not only contains the list of assigned members (`runnable`), but also additional runtime statistics. These statistics include for example which gang member (`curr`) or scheduling gang (`curr_e_task`) is currently executed as well as the number of gang members, which are assigned to this core. Additionally, a reference to the special *idle task*, which executes if there are no gang members assigned to the current core, is also kept in this data structure so that this task is easily accessible if needed. Since the

core-local runqueue is used in the same way as the normal runqueue of the scheduler, this data structure is embedded into the `rq` structure of the Linux kernel. Thereby, no additional management is required for the core-local data structure, as it is necessary for the global schedule gang list.

The data structure itself is again protected by a dedicated lock variable, because forwarding of the global scheduling decision requires shared access to the core-local management data structures[5].

4.3 The Measuring and Accounting of Energy

While the Locked-Out Gang Scheduling algorithm, which the Energy Scheduling Class implements, makes energy measurements possible in a multi-core system in general, the pure scheduling algorithm is still not sufficient to reach the goal of my thesis. To be able to gather energy statistics for individual programs in a time-shared multi-core system, my newly introduced scheduling class has to perform one more task, namely the energy measuring and accounting. As described in Chapter 3, a time-shared system requires that energy is measured and accounted at every context switch between two programs, which share the processor.

Consequently, to be able to determine the energy consumption of individual programs, I had to introduce energy measuring in the scheduling process of the Energy Scheduling Class. As source for the energy measurements I use the RAPL energy counters. With the help of these counters, the energy consumed by a program between two scheduling decisions can be calculated by computing the difference between the energy counters before a measured program gets executed on the processor and after the program is removed from the processor again. Hence, the Energy Scheduling Class has to be able to read and internally save the RAPL energy counter values at every scheduling decision so that it is possible to calculate the consumed energy and account it to the currently executing program.

4.3.1 Measuring the Energy

To provide energy measuring during the scheduling process, I had to be able to internally save the latest RAPL energy counter values. For this purpose, I implemented the data structure as shown in Listing 4.5, which is saved along with each scheduling gang currently handled by the Energy Scheduling Class.

Every time a scheduling class is selected to be scheduled on the processor, the corresponding RAPL counter structure is updated to the latest counter values. This update happens either by reading out the latest counter values from the corresponding MSRs into the data structure [6] or, if a different readout happened just recently for another scheduling gang, by copying over the values of the other gang. This approach ensures that on one side accurate counter values are used for the measurements and on the other side that unnecessary readouts are eliminated. However, if a readout must occur because the last one was too far in the past, the counters are read according to the method explained in Section 3.1.1 to measure short-running

[5] How the propagation of the system-wide scheduling decisions is realized, is explained in detail in Section 4.4.
[6] See Section 2.1.4 for more details about how the values of the RAPL energy counters can be obtained.

```
struct rapl_counter {
    ktime_t last_update;            /* The time point of the last counter readout */

    u32 package;                    /* The values of the different counters */
    u32 dram;
    u32 core;
    u32 gpu;
};
```

Listing 4.5: The data structure used to save latest values of the RAPL energy counters for a scheduling gang.

programs so that it is possible to also determine the energy consumed within short scheduling slices.

When a scheduling gang is removed from the processor, the RAPL counters have to be updated again to be able to calculate the energy consumption for this scheduling slice. For this purpose, the approach for short-running programs is used again if the current scheduling slice length was close to the RAPL counter update rate. Else, the RAPL counters are just read out without waiting for an counter update. This method makes sure that the counter values are as accurate as possible if the program just executes for a short period, since in this case inaccurate values would lead to significant measurement errors. However, if the program runs for a long time, not having as accurate as possible counter values is not crucial for the final energy calculation. By not waiting for an update of the counter values it could happen that the energy consumed during the last counter update and the subsequent readout is lost. But, since the program ran for a long time [7], this missing energy is just a very small portion of the total consumption and consequently not significant for the overall energy consumption of the program. All in all, with this approach, spinning for counter updates within the scheduling process is reduced to only those situations, where they are absolutely necessary for correct measurements.

4.3.1.1 Timer Tick Adaption

For the implementation of Multi-RAPL, I used an additional method to reduce the total amount of time which I need to spin for an update of the energy counters — *timer tick adaption*. The idea is to synchronize the timer ticks at which scheduling decisions happen with the updates of the RAPL counters as shown in Figure 4.2. Thereby, every time the counters have to be read out no looping would be necessary because the counters just have been updated by the processor and hence already contain all the consumed energy of the last update frame.

In the environment of Multi-RAPL this approach was very powerful and could reduce the overall measurement overhead significantly to less than 2%. Unfortunately, in multi-core systems, such as the ones I am targeting at with this thesis, timer tick adaption is not a feasible technique anymore. The problem is, that in a multi-core system each processor core has its own timer tick with its own skew and latency. Hence, constantly synchronizing all timer ticks with the RAPL counter update rate would be too costly and thereby not beneficial enough. An

[7] In my implementation programs must at least run 100 times the average counter update rate so that the calculation error by ignoring the last counter update is below 1%.

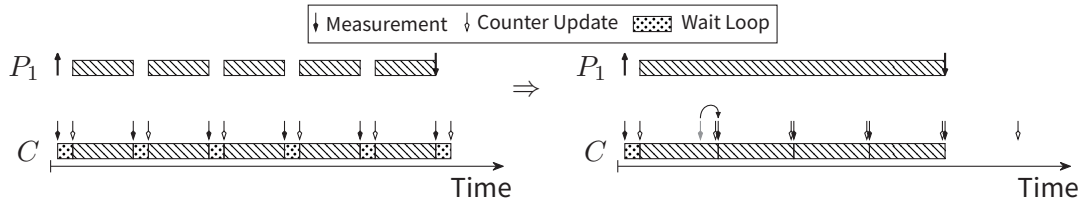


Figure 4.2: Synchronize the scheduler timer tick with the RAPL counter updates so that measuring the program P_1 requires less waiting for accurate energy values.

additional problem in a multi-core system is, that many different locks have to be taken within the scheduler before a final scheduling decision can be made, which further complicates the synchronization process.

Accordingly, I decided not to implement this technique in the Energy Scheduling Class but to only use the already explained methods to reduce the overall number of wait loops in the system.

4.3.2 Accounting the Energy

The accounting of the energy is implemented in the Energy Scheduling Class by accumulating the calculated energy consumption of each scheduling slice at the corresponding leader of the scheduling gang. For this purpose, I extended the `task_struct` of the Linux kernel with the data structure as shown in Listing 4.6.

```
1 struct energy_statistics {
2     u64 uj_package;                 /* Energy reported by the defferent counters */
3     u64 uj_dram;
4     u64 uj_core;
5     u64 uj_gpu;
6
7     u64 nr_updates;                 /* Total number of incremental updates */
8     u64 us_looped;                  /* Total time looped to get accurate values */
9 };
```

Listing 4.6: The data structure used to represent the consumed energy of a pogram.

In order to achieve accurate energy calculations, within this data structure the accumulated energy consumption reported by the different RAPL counters is stored in addition to information about how often these values got updated as well as how much time the Energy Scheduling Class spent during the scheduling process to wait for counter updates.

For the calculation of the consumed energy of each scheduling slice I use the method as explained in Chapter 3 for short-running programs. According to (3.1), the consumed energy is the difference between the two energy counter values minus the energy spent to wait for the counter value update. To determine the energy consumed by the looping, the Energy Scheduling Class is calibrated at every boot of the system. During the calibration, it is determined how much energy is needed if one has to wait for a whole update interval. Based on this value as well as the average update rate, it is possible to calculate the energy consumption during the last scheduling slice as shown in Equation 4.1.

$$E_{\text{slice}} = energy(v_{\text{end}} - v_{\text{begin}}) - \frac{t_{\text{loop}} * \overline{E}_{\text{loop}}}{t_{\text{interval}}} \tag{4.1}$$

E_{slice}	The energy consumed within the current scheduling slice.
$energy$	A function transforming counter values into energy as shown in (2.1).
v_{end}	The energy counter values after the scheduling slice finished.
v_{begin}	The energy counter values before the scheduling slice started.
t_{loop}	The time spent waiting for the counter update.
$\overline{E}_{\text{loop}}$	The average energy consumed to loop for a whole update interval.
t_{interval}	The average update interval length.

In the case that it was not necessary to wait for a counter update because the measurement at the beginning of the scheduling slice happened far in the past, t_{loop} becomes 0 and hence the energy consumption is calculated only as the difference between the two counter values.

4.3.3 Exporting the Energy

While being able to measure and account energy is the main goal of my thesis, I also wanted to provide these measurements to the user of the system. Consequently, I had to add some functionality to the Linux kernel, which enables a normal user program to collect these statistics.

I decided to extend the *procfs* — a special file system usually mounted at `/proc` in a Linux system, which can be used to directly interact with the kernel — with the possibility to read out the accumulated energy consumption of a program. Accordingly, I added a new file into this file system which is available for every executing program. This file contains the current values of the `energy_statistics` structure of the corresponding scheduling task and thereby gives insights into the energy consumed by it.

Of course, this file is only useful if the program is currently measured and hence is managed by the Energy Scheduling Class. Otherwise, the statistics presented in this file do not correlate to the actual energy the program consumes. An overview about which statistics the file contains for a measured process can be found in Listing 4.7.

```
root@measure$ < /proc/100/energystat
package (uJ)        : 1052792342
dram (uJ)           : 54277983
core (uJ)           : 842365434
gpu (uJ)            : 89234
updates             : 303
avg_loop_time (us) : 180
```

Listing 4.7: Example of the energy statistics of a measured program presented in its procfs directory.

Within this file, one can find detailed information about the consumed energy of the program as reported by the four different energy counters as well as the total number of partial energy measurements (`updates`), which happened so far and the average time spent looping for accurate counter values. Hence, if one wants to determine the current energy consumption

of a program, the different energy consumptions presented in this file can be used for this purpose. Furthermore, since the `energystat` file mirrors the internal state of the kernel, this file can even be utilized to monitor the evolution of the energy consumption of a program.

Another file, which I also added to the procfs, namely the `loopstat` file, provides more detailed statistics about the time, which was spent looping for updates of the energy counters. In this file one can find statistics about the overall time spent looping, the total number of loops, the average loop time, as well as a breakdown about how often it has been waited for how long. This file is especially interesting to determine the overhead of the measurement procedure. Listing 4.8 presents the different statistics contained in the file for an example process.

```
root@measure$ < /proc/100/loopstat
loops               : 198
loop_time (us)      : 35640
avg_loop_time (us) : 180

  0 –  99 : 51
100 – 199 : 89
200 – 299 : 37
300 – 399 : 13
400 – 499 : 4
500 – 599 : 2
600 – 699 : 1
700 – 799 : 1
800 – 899 : 0
900 – 999 : 0
    >1000 : 0
```

Listing 4.8: Example of the looping statistics of a measured program presented in its procfs directory.

4.4 The Decision Propagation

A crucial part of the Energy Scheduling Class is, how global decisions about which scheduling gang should run next on the processor are propagated to the individual scheduling instances on each processor core. In this context it is very important, that the Locked-Out Gang Scheduling algorithm requires the scheduling to and from a scheduling gang to happen simultaneously on all processor cores. Otherwise, the necessary property for correct energy measurements cannot be fulfilled. Consequently, I had to implement a fast and robust forwarding mechanism of the global scheduling decisions to the local scheduling instances in the Energy Scheduling Class. To achieve this property, I used a combination of proper locking and remote function execution.

The most important part of this decision scheme is, that every global decision is reliable. For this purpose, I use a shared lock in the Energy Scheduling Class, which ensures that there is only one local scheduling instance at a time, which can make a global scheduling decision. Additionally, I make sure that every decision is deterministic, which means that a different local scheduler would have come to the same result as the one which decided. Another relevant property of a global decision is, that it will not be turned over directly by a subsequent decision

done in the system. This property is necessary because reaching and propagating the decision in the system is a costly operation compared to normal scheduling decisions done in other scheduling classes.

To forward the global decision to the individual local scheduling instances of each processor core, I have to perform two steps. First, the core-local data structures need to be updated according to the decision. For this purpose, I have to insert the assigned gang members into the core-local lists and also set the currently executing scheduling gang in the data structures on all processor cores. Secondly, I have to invoke all local schedulers so that each of them can select a scheduling task corresponding to the global decision. The invocation of the local schedulers is achieved by forcing the main scheduling routine to run on each processor core, which in turn calls the `pick_next_task` method of the Energy Scheduling Class. The `pick_next_task` function can then return the new scheduling task, which should be executed, to the main scheduling routine. While this rescheduling procedure is easy to accomplish locally on the same processor core, activating a local scheduler on a different core is more complicated. To ensure a proper rescheduling of another local scheduler, various locks of the other scheduler instance have to be taken. Unfortunately, since the current local scheduler making the global decision also executes a scheduling routine, all locks of this scheduler instance are currently held as well. Hence, trying to lock the other scheduler instance may result in a deadlock of the system, which must be prevented at all costs.

Consequently, I had to use a different approach to force a rescheduling of the remote schedulers. The solution, which I invented, is to send an *inter processor interrupt* (IPI) to the other processor cores with the result that a special method is called on the other processor core during the handling of the IPI. Within this method, the locks of the local scheduler instance are taken and the core scheduler routine is activated. To prevent sending unnecessary IPIs in the system[8], I do not just invoke a special method on the other processor core, but also internally monitor this operation. Thereby, I can avoid sending further IPIs, if the one, which just has been sent, is not yet processed on the remote processor. Instead, I adapt the already sent one in such a way, that the very latest operation will be executed during the handling of the IPI. This technique ensures, that always the most recent remote rescheduling request is executed without unnecessarily flooding the whole system with IPIs.

4.5 The Class Time-Sharing

As described earlier in Chapter 3, the integration of the Energy Scheduling Class with a higher priority than the scheduling class for normal user level programs is problematic for the responsiveness and usability of the system. Within the main scheduling routine, scheduling decisions are put into effect according to the priorities of the corresponding scheduling classes. Hence, if the Energy Scheduling Class wants to run a scheduling gang, all the other scheduling classes with a lower priority are not allowed to run anything in the whole system, because the Energy Scheduling Class has a higher priority and additionally occupies the complete processor. Accordingly, long-running programs, which are scheduled by the Energy Scheduling Class, will cause all the other programs managed by the Completely Fair Scheduling Class to starve with the consequence that the system itself is not usable anymore.

[8] Inter processor interrupts are an expensive operation in a multi-core system.

However, I wanted my new energy measuring functionality to be usable in the background with the remaining parts of the system working as usual. Correspondingly, I had to introduce a functionality into the Energy Scheduling Class which allows the system to remain usable although there are long-running measured programs. The idea, which I implemented to maintain the system's usability, is to give up the scheduling from time to time so that lower priority scheduling classes are able to run their tasks. Consequently, I do not only time-share the processor between the scheduling tasks of the Energy Scheduling Class but also between scheduling tasks of other classes.

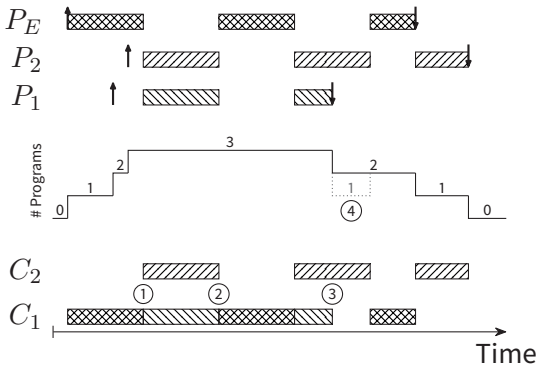


Figure 4.3: Implementation of the scheduling class time-sharing algorithm. The figure shows an example where the processor is shared between one measured program P_E and two normal programs P_1 and P_2.

For this reason, I only allow the Energy Scheduling Class to run continuously for a specific amount of time, if there are other scheduling tasks available in the system than the ones managed by my class. This time period is determined based on the total number of tasks belonging to all the scheduling gangs managed by the Energy Scheduling Class. Accordingly, within the `pick_next_task` method I check constantly if the calculated time frame for the Energy Scheduling Class is depleted when there are additional scheduling tasks in the system. If the current time frame is exhausted, scheduling is given up to allow the other tasks to run as well, as illustrated in Figure 4.3 at number (1). When the other tasks finish or if they executed long enough[9], the Energy Scheduling Class continues scheduling its tasks (number (2) in Figure 4.3).

With this approach neither long-running tasks in the Energy Scheduling Class nor in any lower priority class can prevent the execution of scheduling tasks in the other classes. Additionally, as both time slices are calculated in the same way, the processor is shared equally among all the tasks in the system.

Unfortunately, making the Energy Scheduling Class give up the whole scheduling process in the described way has some difficulties. It could happen that, although there are multiple normal tasks available to run in the system, some processor cores do not have a normal task assigned to them. But since the Energy Scheduling Class can only give up the entire processor, these unoccupied processor cores have to run another task, which does not belong to the

[9] Whether the other tasks executed long enough or not is determined based on the total number of scheduling tasks in the system.

Energy Scheduling Class. In such a situation, when a processor core has nothing to run, the Linux kernel uses a special idle task, which is executed on the corresponding core. Unfortunately, before the idle task is allowed to run, the main scheduling routine checks whether there are still other higher-priority scheduling tasks available on the processor core, which can be executed instead of the idle task. In Figure 4.3 at number (3) a similar situation is visualized. After the program P_1 finished its execution, the idle task has to run on core C_1 to fulfill the properties of the Locked-Out Gang Scheduling algorithm but the main scheduling routine prefers to schedule the also available measured program P_E since it has a higher priority. Consequently, the Energy Scheduling Class has to make the main scheduling routine think that there are no scheduling gang members available on the processor core although there are gang members available, which just should not be executed. To accomplish this functionality, the Energy Scheduling Class determines whether it is currently giving up scheduling in favor of other scheduling classes and adapts the internal number of available scheduling tasks accordingly (as it can be seen in Figure 4.3 at number (4)) so that the tasks managed by the Energy Scheduling Class are not counted by the main scheduling routine. At the point in time when the Energy Scheduling Class continues its scheduling process again, this adaption is inverted.

4.6 Higher Priority Tasks

An additional problem for the Energy Scheduling Class is the presence of other higher priority tasks in the system. As explained earlier, the Energy Scheduling Class' priority is settled between the Real-Time and the Completely Fair Scheduling Classes. Consequently, there can be other scheduling tasks in the system, which have a higher priority and thereby are allowed to run in favor of scheduling gang members. While *real-time* and *deadline* scheduling tasks are very seldom in a normal system, tasks belonging to the Stop Scheduling Class, which has the highest available priority, occur from time to time, because important kernel functionality like the migration of scheduling tasks between processor cores is executed with the priority of this scheduling class.

The problem for the Energy Scheduling Class with these higher priority tasks is, that these tasks are allowed to replace a scheduling gang member on any processor core and thereby violate the main property of the Locked-Out Gang Scheduling algorithm, namely that only scheduling tasks, which belong to the same scheduling gang, are allowed to run simultaneously on the processor. The proper way to handle such violations is to stop the execution of the entire scheduling gang if the replacement of one of its gang members is detected.

Unfortunately, until now the current implementation of the Energy Scheduling Class does not handle these violations properly. Instead, the gang member is simply executed again as soon as the higher priority task finishes. However, since only scheduling tasks belonging to the Stop Scheduling Class, which normally run for a very short time period, are common in average user systems, this inconsistent behavior does not influence the overall functionality of the Energy Scheduling Class significantly. Still, I plan to implement support for such scenarios in future versions of the scheduling class so that the other higher priority classes like the Real-Time or the Deadline Scheduling Class can be used together with energy measurements.

5 Evaluation

While the previous chapters introduced the design and the implementation of the Energy Scheduling Class, which allows energy measurements of individual programs in a multi-core system, I would like to concentrate on the evaluation of this new energy measurement facility in this chapter. For this purpose, I want to determine the measurement accuracy as well as the measurement overhead of the Energy Scheduling Class. Accordingly, I perform multiple experiments where I use the Energy Scheduling Class to measure the energy consumption of varios benchmarks in different situations. The following sections will show and discuss the results of these experiments.

5.1 Measurement Setup

All the measurements presented in the next sections have been performed on a quad-core Intel® Haswell Core™ i7-4770 machine with 3.40 GHz and 8 GB of physical memory. The *Hyper-Threading* as well as the *Turbo-Boost* processor features were disabled during the whole evaluation process so that possible influences of these performance boosting techniques could be eliminated from the final measurement results. On the machine I executed a 64-Bit Linux kernel version 4.2.3 with my new Energy Scheduling Class integrated.

Since the aim of this thesis is to evaluate a multi-core system, I chose benchmarks from the *NAS Parallel Benchmark* suite [24] to determine the overhead and accuracy of my new measurement technique because all benchmarks contained in this suite are capable to spread to multiple processor cores in the system. While the suite provides different possibilities to achieve this internal parallelism in the benchmarks, I decided to use the *OpenMP* [25] version of it since this version uses threads to accomplish the parallel execution of the benchmarks which nicely fits into my measuring model.

The benchmark suite contains 10 different benchmarks, which are mostly CPU and memory intensive. Only the *Data Cube* benchmark (dc) of the suite is IO intensive. Each benchmark exists in multiple different problem sizes, which were introduced by the creators of the suite so that the runtime of the individual benchmarks can be adapted to a reasonable value on the corresponding measurement system.

For my evaluation I chose the benchmarks *bt*, *lu*, *sp*, and *ua* with the problem size *A*, the benchmarks *cg*, *ep*, and *ft* with the problem size *B*, and the benchmarks *is* and *mg* with the problem size *C* so that each benchmark runs for at least 20 s in their not parallelized version. Thereby, all the benchmarks still have a feasible runtime even if they are executed with the maximal available parallelism in the system. Unfortunately, I could not use the *dc* benchmark for my evaluation, because, due to its IO intensive characteristic, the benchmark self-suspends often to wait for the hard drive, which prevents proper baseline measurements [1] as outlined

[1] See Section 5.2 for an explanation how the baseline measurements are taken for my evaluation.

in Section 5.6.

To determine the influences of my measurement method on programs with different types of internal parallelism, I executed all the analyzed benchmarks with one, two, three, and four threads. Thereby, I am able to gather results for various load situations of the system and accordingly have more insight into the overall behavior of my new energy measurement technique.

5.2 Baseline Measurements

Before being able to do any evaluation of the accuracy and overhead of my new scheduling class, I needed to determine the normal characteristics of all the measured benchmarks. For this purpose, I executed all the chosen benchmarks in a stripped down Linux environment, where only the kernel itself was running together with the benchmarks. During the execution, I measured the time each benchmark was scheduled on the processor (*CPU time*), the time which each benchmark took from start until termination (*wall-clock time*), and the energy each benchmark consumed.

To calculate the energy, I read the RAPL energy counters before the benchmark started and after it finished and then computed the difference between these two values. Since nothing else was running in the system and hence no processor-sharing happened, the computed energy completely belongs to the executed benchmark. To read out the energy values at the beginning and at the termination of the program, I used the same technique as the Energy Scheduling Class internally. The start of the benchmark is synchronized with a RAPL counter update and after the termination, I wait for an update only if the program did not execute long enough. Since the measurements done in the Energy Scheduling Class use all available energy counters, the baseline measurements also contain the energy consumptions reported by all four RAPL energy counters. Hence, in addition to the two different runtimes, the baseline measurements include the energy reported by the PKG energy counter (*package*), by the PP0 energy counter (*core*), by the PP1 energy counter (*gpu*), and by the DRAM energy counter (*dram*).

To have statistically evaluable results, each measurement is repeated 20 times. Thereby, outliers, which may occur during the measurements, can be correctly treated during the final evaluation of the results.

Figure 5.1 presents the results for all benchmarks included in the baseline measurements. The figure shows the average over 20 executions for every benchmark, each executed with one to four threads. The standard deviation for all measurements is below one percent and is therefore not shown in the plots. All six different measurement criteria are shown separately in their own graph.

The graphs show, that most benchmarks scale very well in their performance from one to four threads. The wall-clock time (Fig. 5.1a) of those benchmarks reduces according to the number of threads. Only the mg.C benchmark scales comparatively bad. Its wall-clock time decreases just slightly with the increase to three or four threads. The reason why this benchmark does not scale well can be found in Figure 5.1b. This graph shows the time all threads of the benchmarks were executed on the processor in sum. The mg.C benchmark's CPU time increases with an increasing number of threads, which means that the threads of

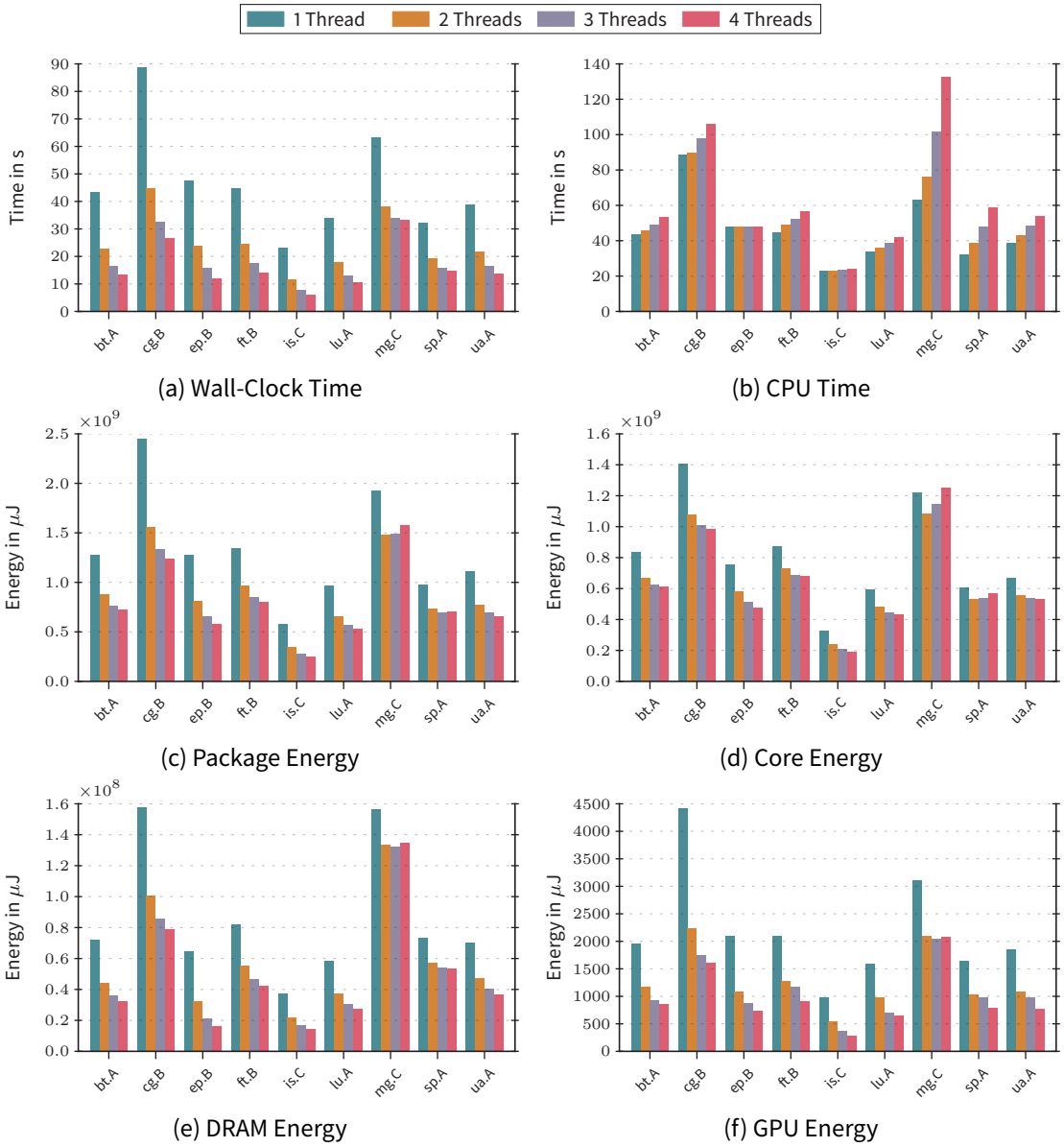


Figure 5.1: Results of the six measurement criteria of the baseline measurements. Each bar in the plots represents the average of the 20 repetitions. Error bars are not included in the plots since the standard deviation for all results is below one percent.

the benchmark interfere with each other during the execution. Other benchmarks such as ep.B or is.C have nearly equal CPU times for the different number of threads, which shows that their threads do not hinder each other, which in turn results in a reduced wall-clock time.

The energy consumption measured for the different benchmarks correlates most of the time with their wall-clock time. Especially the energy dissipated by the whole CPU with all its components, which is reported by the PKG energy counter (Fig. 5.1c), matches the corresponding wall-clock time very well. However, since the processor is stressed more with an increasing number of threads used by the benchmarks, the energy does not decrease with an increasing thread number as much as the corresponding wall-clock time. The mg.C benchmark, which has only a small runtime decrease from three to four threads, shows even an energy increase for this constellation. The same can be noted for the sp.A benchmark as well. The reason for this package energy increase can be found in the graph for the energy reported by the PP0 energy counter (Fig. 5.1d), which shows the core energy consumption of the benchmarks. In this graph one can see significant core energy increases with an increasing number of threads for the mg.C benchmark as well as a slight increase for the sp.A benchmark. In general this graph visualized the influence of the usage of multiple threads on the energy consumed by the processor cores very well. With the increase from three to four threads the core energy only changes slightly for most benchmarks although the wall-clock time decreases noticeably with the additional thread.

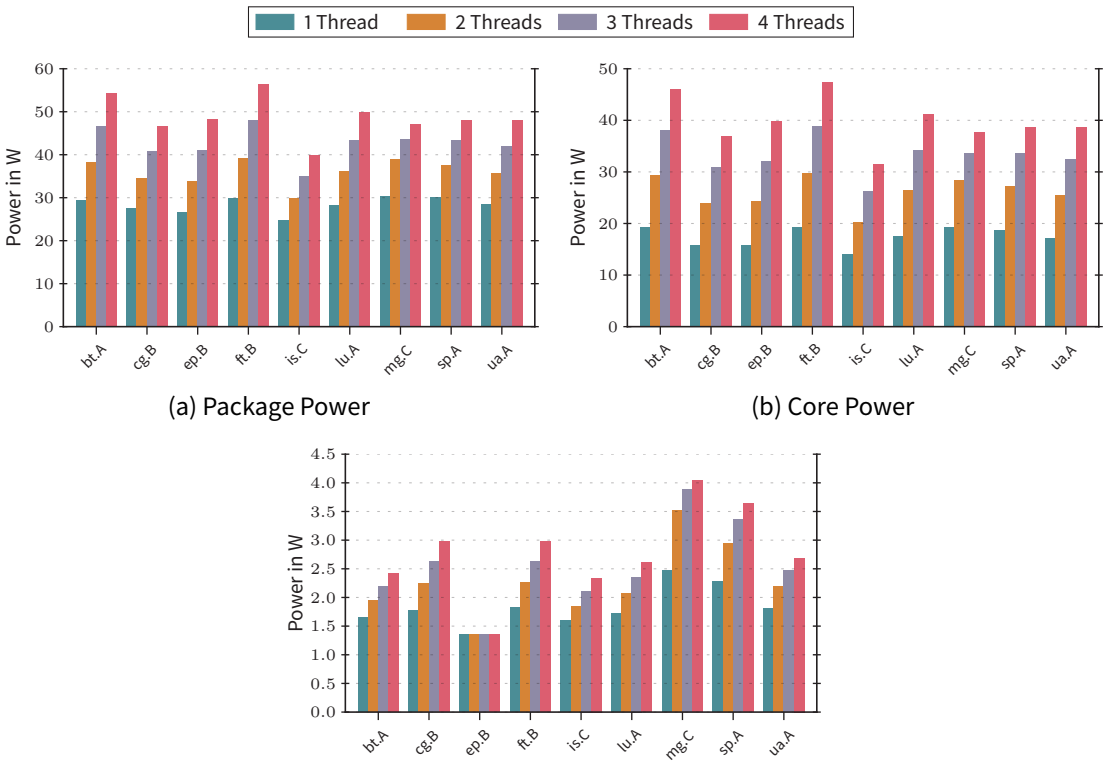
The graph containing the DRAM energy (Fig. 5.1e) visualized another interesting property, namely that the benchmarks can have different power characteristics. For example, the mg.C benchmark has an equal or much higher DRAM energy consumption than the cg.B benchmark although it executes shorter with one and two threads or just a little bit longer with three and four threads. This behavior is also visible with the ep.B and ft.B benchmarks. Figure 5.2, which shows the average power consumption of all benchmarks, explains this property very well. The mg.C benchmark has a significantly higher average DRAM power consumption than the cg.B benchmark.

Similar examples, where equally long running benchmarks consume noticeably different energy, can also be found for the package or core energy although for these energy measurements the relative differences in the average power consumption are smaller than with the DRAM energy measurements.

The energy reported by the PP1 energy counter, which correlates to the GPU energy (Fig 5.1f) is very similar to the wall-clock time graph. The reason for this behavior is, that the benchmarks themselves do not use the GPU at all but only the kernel uses the GPU when it updates the console output. Hence, the energy reported by these counters is the energy used to update the screen during the execution of the benchmark. Consequently, I will not continue evaluating this energy counter during the remaining measurements.

5.3 Solo Measurements

As a first experiment to determine the accuracy and overhead of my new energy measurement facility for multi-core systems, I executed all benchmarks with one to four threads as *measured programs* solely in the system. Accordingly, every benchmark with its configured number of threads was scheduled by the Energy Scheduling Class. In parallel to the measured program



(a) Package Power

(b) Core Power

(c) DRAM Power

Figure 5.2: Average power consumption of all benchmarks calculated from the average energy reported by the different RAPL energy counters and the total runtime of the benchmarks. The average power consumption of the GPU is left out, since the energy reported by this energy counter is not consumed by the benchmarks itself.

nothing else was running in the system, which significantly consumed energy or occupied the processor.

For statistical significance, I repeated every combination of number of threads and benchmarks 20 times. Thereby, I was again able to properly account outliers in the results if they exist at all.

Since energy was measured in this experiment within the kernel, I only had to collect the final energy statistics from the *procfs* directory of the benchmark as described in Section 4.3.3. In addition to the energy consumption, I also measured the time the benchmark was scheduled on the processor (CPU time) as well as the time the benchmark run in total (wall-clock time). Accordingly, this experiment contains the same measurement criteria as the baseline measurements.

With this measurement experiment, I wanted to identify the performance overhead as well as the measurement accuracy of my new energy measurement facility when it operates solely in the system. In the optimal case, all benchmarks should take the same wall-clock time and CPU time as in the baseline measurements, since there should be no overhead in the Energy Scheduling Class when the system is without load. Furthermore, the measured

energy consumption should be similar to the ones done during the baseline measurements. A slightly lower energy consumption would be an even better result, because my new measurement method is more accurate than the one used for the baseline measurements. For example, the Energy Scheduling Class does not account energy consumed by kernel tasks or other background load to the measured program as it was done incorrectly by the baseline measurements.

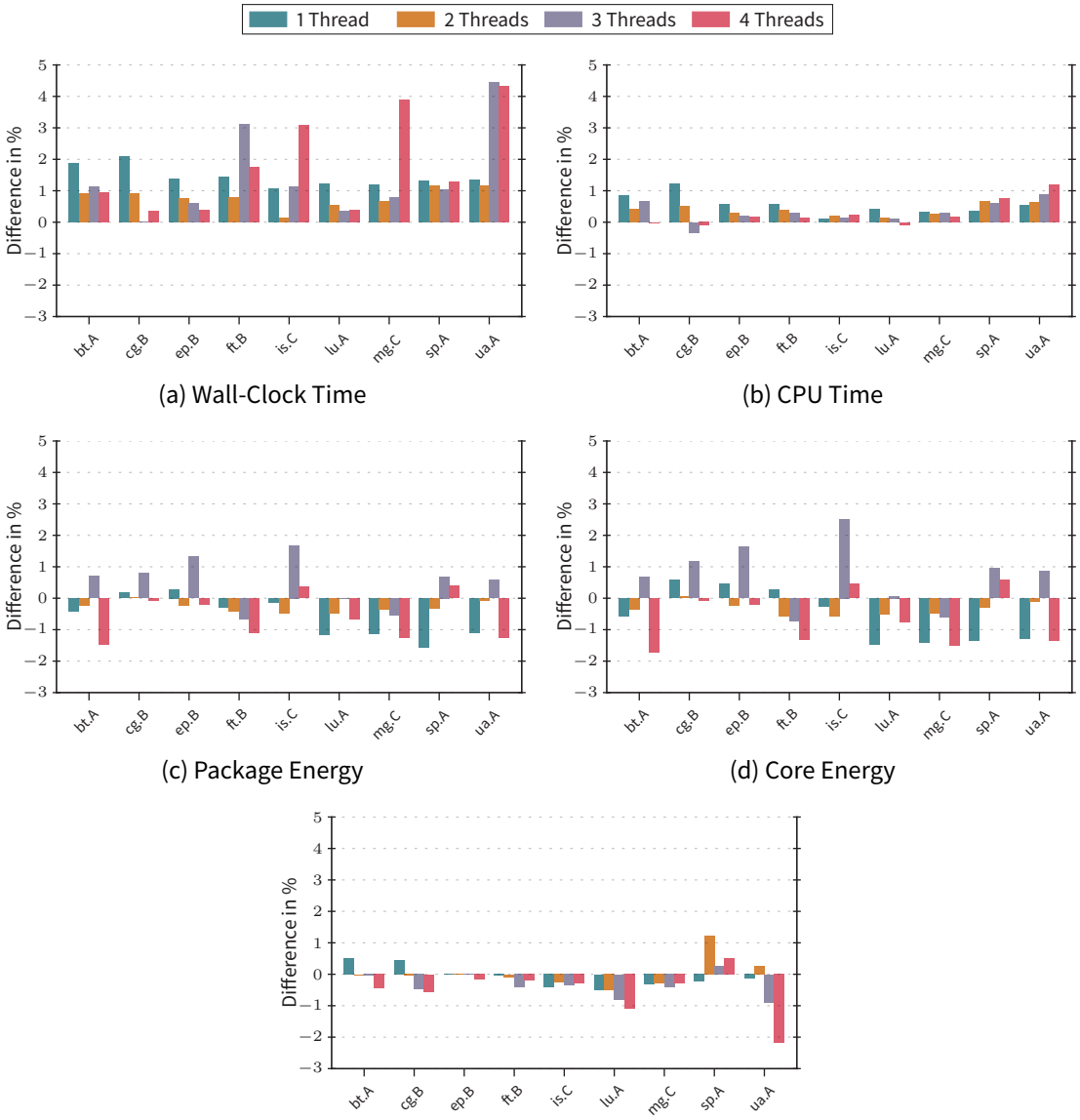
In Figure 5.3 the results of this experiment are visualized. The values shown in the graph are the difference in percent between the measurements done within this experiment and the corresponding baseline measurements. This way of visualization allows identifying the performance overhead as well as the measurement accuracy very easily. The standard deviation of the values given in the graphs is not shown again because it is for all of them below two percent.

As it can be seen in the figure, the wall-clock time difference (Fig. 5.3a) of the various benchmarks with the different number of threads is for all of them below five percent. This overhead is compared to other measurement facilities for multi-core systems such as LEA^{2}P [29], which also has a runtime overhead of up to five percent, an acceptable result. For most benchmarks, the runtime overhead is even below two percent, which I consider a not noticeable overhead for an average user. The reason that there is, in contrast to the expectation, an overhead measurable in the wall-clock time compared to the baseline measurements, is that background load like kernel threads for file system management or system logging facilities cannot run in parallel to the measured program. During the baseline measurements, where the benchmarks were scheduled by the Completely Fair Scheduling Class, such background load could run in parallel to the benchmarks without increasing its wall-clock time significantly, especially if the benchmarks did not use all cores of the processor. With the Energy Scheduling Class occupying the whole processor, running such background tasks requires that the processor is completely given up, which in turn needs a full descheduling of the currently measured program in favor of the background task. Accordingly, the overall runtime of each benchmark increases, since the benchmark can not execute if there are some background tasks running in the system.

In the CPU time measurement results of the benchmarks (Fig. 5.3b), which differ from the baseline measurements at the most about one percent, one can see that the energy measurement facility itself does not influence the measured program significantly. The small difference which still exists for some benchmarks can be accounted to the energy measuring technique used by the Energy scheduling class[2]. The time spent during the energy measuring to ensure accurate energy measurements is added by the kernel to the time the measured program is scheduled on the processor and hence increases the final CPU time of the benchmarks. Accordingly, the more partial energy measurements are necessary to determine the total energy consumption of a program, the more increases the final CPU time of the program.

The graphs containing the different energy consumptions reported by the three energy counters (Fig. 5.3c-e) show, that my new energy accounting technique for multi-core systems works reliably if it is used solely in the system. The energy measured by the Energy Scheduling Class internally differs most of the time by less than two percent from the baseline measure-

[2] See Section 4.3 for more details about how energy measuring and accounting is implemented in the Energy Scheduling Class.



(a) Wall-Clock Time

(b) CPU Time

(c) Package Energy

(d) Core Energy

(e) DRAM Energy

Figure 5.3: Results of the five measurement criteria of the solo measurements. Each bar in the plot visualizes the difference between the result of this measurement and the corresponding one of the baseline measurement.

ments, which is a high accuracy similar to other measuring techniques like LEA^{2}P [29], which reaches a measurement accuracy of up to 96 percent. Only the is.C benchmark executed with three threads reports 2.51 % more core energy consumption and the ua.A benchmark run with four threads consumes 2.18 % less DRAM energy.

For most of the benchmarks my new energy measuring facility reports lower consumed energy than determined in the baseline measurements. This property aligns nicely with my expectation, since the Energy Scheduling Class now properly accounts energy consumed by background tasks, which was not correctly done by the baseline measuring method. As already mentioned previously, the baseline measurements contain all the energy consumed during the whole execution time of the benchmark. Accordingly, this energy consumption may also contain energy of other programs such as kernel tasks, which were scheduled together with the benchmark by the Completely Fair Scheduling Class.

To sum it up, this experiment shows that the Energy Scheduling Class can be used to make accurate and low-overhead energy measurements of solely running programs in a normal Linux environment.

5.4 Background Measurements

Within the second experiment for my new energy measuring technique for multi-core systems, I wanted to determine how accurate and usable this technique is, if its measured programs not only run solely in the system but together with some other programs. For this purpose, I executed all benchmarks with one to four threads as *measured programs* together with another program performing a simple busy loop as background load. Accordingly, the benchmark was managed by the Energy Scheduling Class while the busy-loop program was scheduled by the Completely Fair Scheduling Class.

To be able to properly treat ouliers within the measured data, every combination of benchmarks and number of threads was repeated 20 times. During the experiment, the busy-loop program was started together with the benchmark and killed as soon as the benchmark finished its task. As measurement criteria I used, similarly to the previous measurements, the energy consumptions reported by the three energy counters (package, core, and DRAM) as provided by the Energy Scheduling Class in the *procfs* file system and the time each benchmark was scheduled on the processor (CPU time). The time each benchmark required to fulfill its tasks (wall-clock time) could not be compared to the baseline measurements anymore in this experiment, because the presence of the additional load program influenced the overall runtime of the benchmark. The processor had to be shared between the benchmark and the busy-loop program, which resulted in significant overall runtime increases.

As the aim of this experiment was to identify the measurement overhead and accuracy of my new energy measuring method in an environment where its measured programs do not run solely anymore but in parallel to other not-measured programs, the expectations at the experiment results were in general similar to the ones for the previous experiment. Especially, because the not-measured busy-loop program should not influence the measured benchmarks in their general behavior, the different energy consumptions as well as the CPU time should be comparable to the previously measured values.

In Figure 5.4 the measurement results of this experiment are presented. Since I wanted to

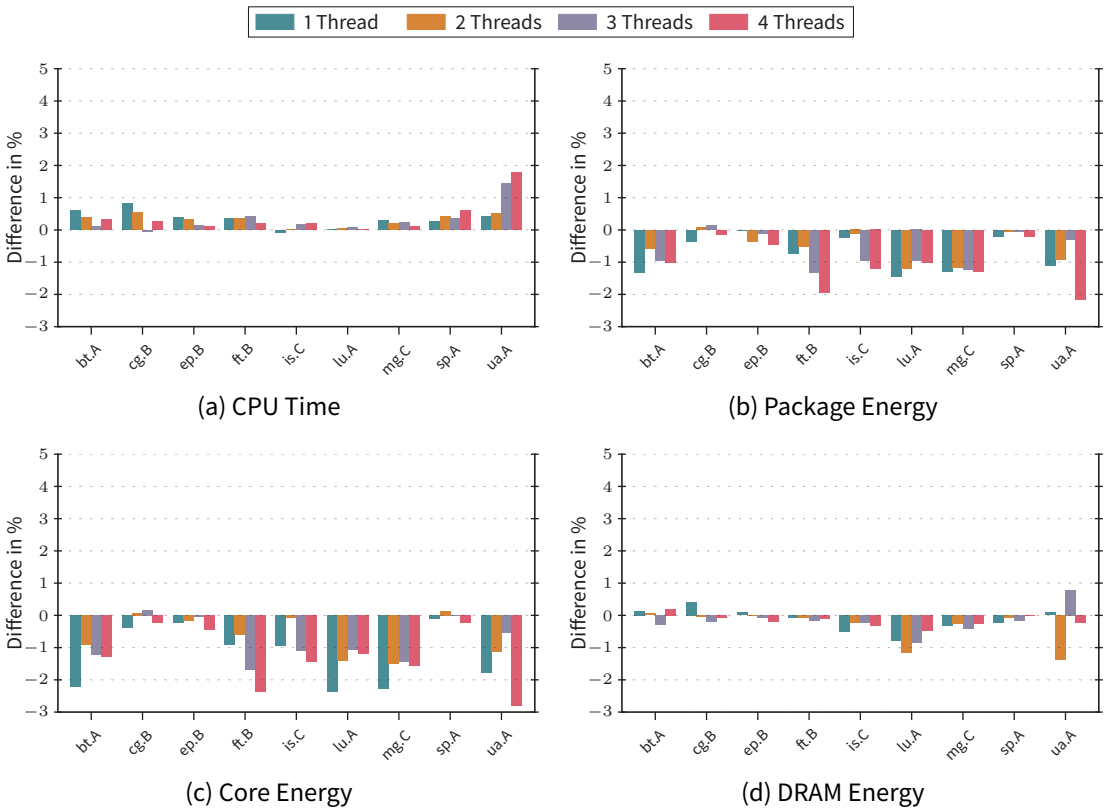
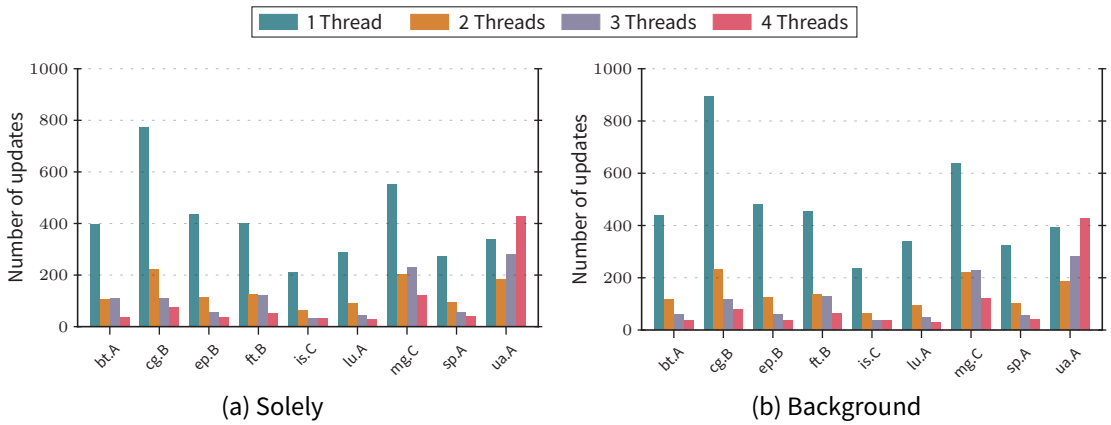


Figure 5.4: Results of the four measurement criteria of the background measurements. Each bar in the plot visualizes the difference between the result of this measurement and the corresponding one of the baseline measurement.

compare the measured values against the corresponding ones from the baseline measurements, the final values are presented in the graphs, similarly to the previous experiment, as relative differences to the corresponding baseline measurement values. This way of presenting the results allows identifying the accuracy and overhead of my measurement method easily.

The measurement results for the CPU time of the different benchmarks (Fig. 5.4a) indicate, that the parallel running background load does not influence the overall runtime of the benchmark significantly. The Energy Scheduling Class is still able to measure the energy consumption of the benchmarks without any noticeable increase in the runtime of the benchmarks. Only for the ua.A benchmark a small increase of the CPU time can be noted, which raises further when more threads are used by the benchmark. The reason for this behavior can be found in the blocking synchronization technique used by the benchmark. Hence, if the benchmark uses more threads, it has to block more often to synchronize all threads, which each require a scheduling switch on the corresponding processor cores. If the program is scheduled by the Energy Scheduling Class, blocking inside the program may additionally result in a complete descheduling of it, which in turn necessitates an energy measurement. Accordingly, the ua.A benchmark requires more partial energy measurements in the kernel with an increasing number of threads, which lengthens the overall CPU time of the benchmark.



(a) Solely (b) Background

Figure 5.5: The number of partial energy measurement done by the Energy Scheduling Class during the execution of the benchmarks while they run solely in the system or together with other programs.

As it can be seen in Figure 5.5, this behavior can only be noted for the ua.A benchmark. All the other benchmarks require less partial energy measurements with an increasing number of threads, since they run shorter and have longer scheduling slices.

The other graphs, which contain the different energy consumptions reported by the RAPL energy counters (Fig 5.4b-d), present a very similar result as the corresponding ones of the previous experiment. For most benchmarks the Energy Scheduling Class determines a lower total energy consumption than measured during the baseline measurements. This outcome originates in the measurement technique used to gather the baseline measurements. The baseline measurement technique assumes, that all the energy consumed from the start of the benchmark to the termination of the benchmark, belongs to the benchmark itself. Unfortunately, since it could happen that kernel activities are executed together with the benchmarks, this assumption is not correct. Hence, the baseline measurements may report more energy than actually consumed by the benchmarks. The measurement technique used by the Energy Scheduling Class, on the other hand, handles such scenarios correctly and only reports the actual energy consumed by the measured program. All in all, the difference between the energy consumptions determined during this experiment and the baseline measurements is below three percent which is still a high overall measuring accuracy.

However, by comparing the results of this experiment with the ones of the last experiment it can be noticed that the difference to the baseline measurements increased for some benchmarks. A reason for this increase can be, that the RAPL energy counters are not as accurate as expected but mix the energy characteristic of the busy loop (24.2 W average package power consumption and 12.5 W average core power consumption) with the energy characteristic of the benchmarks. This behavior would also explain why the difference between the results is larger for benchmarks with a high average power consumption such as ft.B or mg.C.

In total, the experiment could show that my new energy measurement technique for multi-core systems can also be used in an environment where measured and not measured programs run in parallel. The low runtime overhead as well as the high measurement accuracy are not influenced significantly by the continuous background load in the system.

5.5 Concurrent Measurements

The third experiment, which I used for the evaluation of my energy measurement facility for multi-core systems, has the aim to determine the measuring accuracy as well as the runtime overhead of the Energy Scheduling Class if it is used to measure multiple programs in parallel. Furthermore, with this experiment I also wanted to find out whether parallel measurements of programs with different energy characteristics influence each other significantly.

For this purpose, I executed all benchmarks with one to four threads together with all other benchmarks with the same number of threads as *measured programs*. Accordingly, both benchmarks were managed and scheduled by the Energy Scheduling Class and hence got their energy measured while they executed. Similar to the previous experiments, I collected statistics about the package, core, and DRAM energy as well as the time each benchmark was scheduled on the processor (CPU time). Measuring the time which each benchmark needed to complete its tasks (wall-clock time) was not feasible, similar to the previous experiment, since running multiple benchmarks in parallel requires that the processor must be multiplexed between them, which in turn lengthens the overall runtime of each individual benchmark.

To have results, which can be compared to the ones done during the baseline measurements, this experiment was organized as follows. The two selected benchmarks were started simultaneously. If a benchmark finished, it was restarted so that the not yet finished benchmark continuously runs together with another program. As soon as both benchmarks were at least restarted 20 times, both programs were killed. This method ensured, that there were always two benchmarks running in parallel and that all benchmarks had at least 20 repetitions, which makes a proper evaluation possible.

Before I started the measurements for this experiment, I performed additional measurements, during which I determined the last-level cache (LLC) misses of all benchmarks with the *perf* tool [19] to identify memory intensive benchmarks. For this purpose, I executed each benchmark managed by the Completely Fair Scheduling Class solely as well as concurrently to itself and measured the LLC misses. In addition, I run the benchmarks again solely and concurrently to itself, but this time managed by the Energy Scheduling Class. The measurements revealed that a side product of the Locked-Out Gang Scheduling algorithm used by the Energy Scheduling Class is, that the benchmarks do not influence each other significantly. The long scheduling slices in combination with the gang-scheduling of the benchmark threads and the occupying of the whole processor made the benchmarks behave as if they were executed solely in the system. Such a behavior could not be seen when the benchmarks were managed by the Completely Fair Scheduling Class.

Accordingly, the expectation for the results of this experiment were again similar to the ones for the previous experiments. Energy should still be measured and accounted accurately to the corresponding programs and the overhead occurring because of the energy measurements should be small. However, since programs with different energy characteristics[3] were measured in parallel, larger measurement errors could occur depending on the accuracy of the underlying energy counters.

Figure 5.6 presents the results of this experiment. Every subfigure contains the results of one measurement criteria whereas each plot in the subfigures shows those results, when the

[3] See Figure 5.2 for the average power consumptions of the different benchmarks.

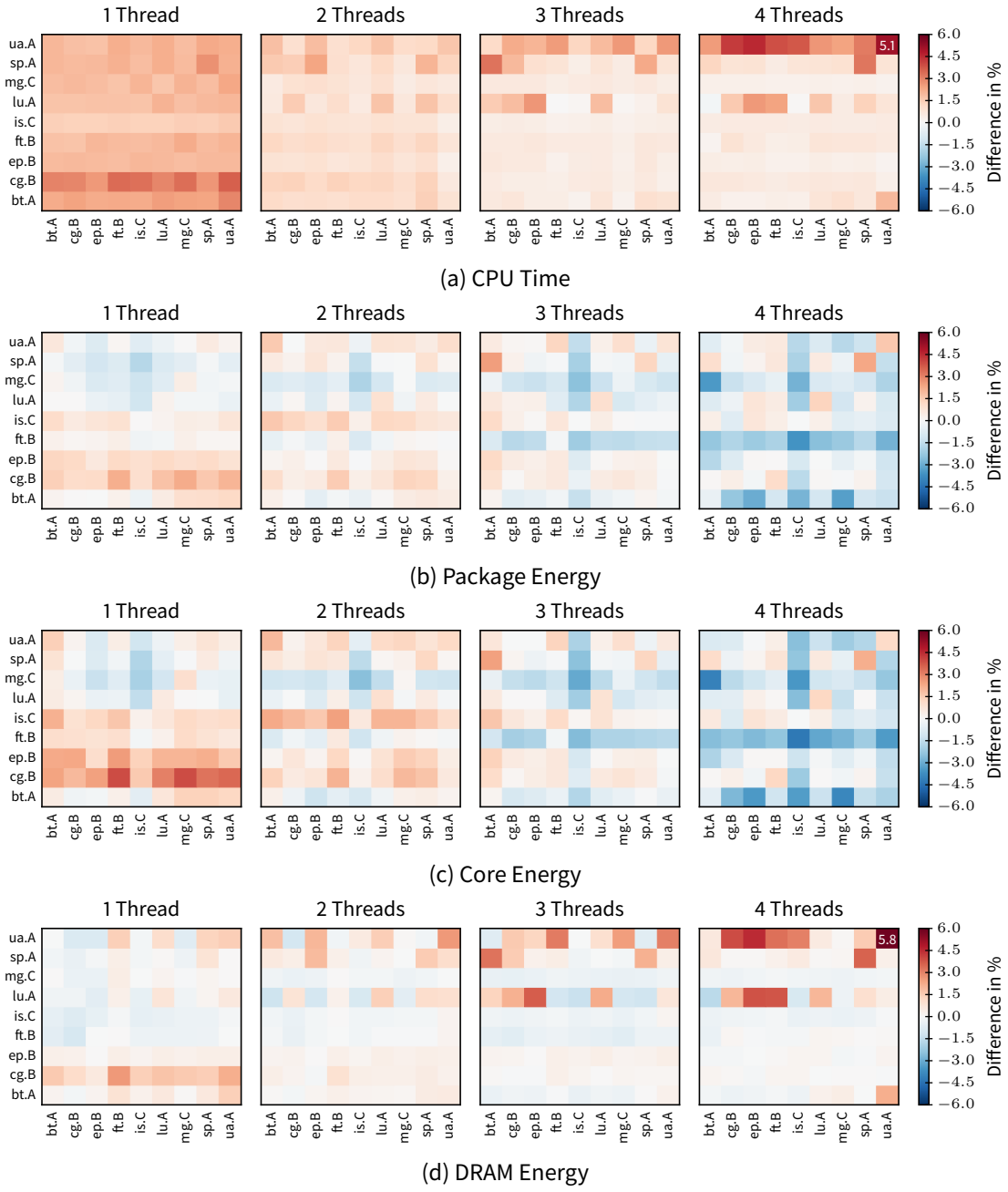


Figure 5.6: The results of the four measurement criteria of the concurrent measurements. Every cell in the plots visualizes the difference between the baseline measurement of the benchmark mentioned at the row and the concurrent measurement of this benchmark when it runs together with the benchmark mentioned at the column. Above the plots it is stated with how many threads each benchmark was executed.

benchmarks were executed with the number of threads stated above the plot. Within the plot, each cell visualizes the difference between the baseline measurements and the ones done during this experiment of the benchmark mentioned at the current row, when it runs in parallel to the benchmark mentioned at the current column. For example, the second cell in the lowest row of a plot visualizes the difference between the measurements taken during this experiment for the bt.A benchmark when it runs in parallel with the cg.B benchmark and the corresponding baseline measurements of the bt.A benchmark.

By looking at the results for the CPU time of the various benchmarks (Fig. 5.6a) it can be seen, that in general the runtime overhead increased compared to the previous experiments. While the other experiments most of the time showed CPU time increases only up to two percent, within this experiment the runtime is raised up to five percent in the worst case. The highest overhead for all benchmarks can be seen when they are executed with only one thread each. The reason for this could be that scheduling switches between the two parallel running instances happen often, which every time requires an energy measurement. During the energy measurements, from time to time it is necessary, that energy counter updates must be waited-for, which in turn lengthens the overall execution time of the benchmarks. This behavior also explains why the overhead gets smaller with an increasing number of threads since the number of threads is used in the Energy Scheduling Class to calculate the scheduling slice length. The more threads a program has, the longer it is allowed to run continuously, which means that scheduling switches and thereby energy measurements happen less often. Only the ua.A benchmark shows a different behavior. Its CPU time gets worse with in increasing number of threads. Since a similar behavior could already be seen during the last experiments my assumption is that this increase happens not because the program executes in parallel to others, but because of the internal synchronization mechanism of the benchmark. As it can be seen in Figure 5.5, which visualizes the number of partial energy measurements performed by the Energy Scheduling Class, the ua.A benchmark requires more measurements with an increasing number of threads in contrast to the other benchmarks. As already explained during the last section, the reason for this increase of partial energy measurements for the ua.A benchmark is located in the blocking synchronization technique used by the benchmark. The more threads are used by the benchmark, the more often the benchmark has to block, which in turn requires more scheduling switches and thereby more partial energy measurements in the kernel, which lengthens the overall runtime of the benchmark. Still, all in all the overhead introduced by the Energy Scheduling Class to measure the energy consumption of the benchmarks when they are executed in parallel to other measured benchmarks is small. Other measurement techniques like LEA^{2}P already have an overhead of up to four percent if they only have to determine the energy consumption of one program in the system.

When looking at the energy statistics for the different benchmarks gathered during this experiment (Fig. 5.6b-d) a slightly different picture arises. The final values are comparable to the ones collected during the baseline measurements. Energy increases as well as decreases with values up to five percent can be found in the plots. However, starting with three threads, the differences in the energy characteristics of the benchmarks, as shown in Figure 5.2, become visible in the results of this experiment as well. For example, in the core or package energy plot with four threads it can be seen, that the ft.B benchmark having the highest average power consumption constantly gets lower energy accounted than before. Furthermore, every time

when a benchmark runs together with the is.C benchmark, which has the lowest average power consumption, the final package energy consumption of the benchmark decreases. A reason for this behavior could be, that the underlying energy counters used for the measurements are not accurate enough and hence mix the two different energy characteristics, which was also noticed during the previous experiment.

The plot for the DRAM energy measurements does not show such patterns. Its values correlate with the CPU time of the benchmarks. Every time a benchmark runs longer than during its baseline measurements its DRAM energy consumption raises as well. However, since the measurements fort he last-level cache misses do not show a similar pattern, the reason for this behavior must be an accounting and measuring problem in the Energy Scheduling Class or an inconsistency in the behavior of the corresponding RAPL energy counter.

Altogether, this experiment shows, similar to the previous ones, that my new energy measurement technique for multi-core systems has a high accuracy and a low runtime overhead. Even measuring multiple programs in parallel with different energy characteristics does not influence the final energy consumption of each program significantly.

5.6 Interactive Programs

While all measurements presented in the last sections exclusively used the programs from the NAS Parallel Benchmark suite to evaluate my new energy measurement facility for multi-core systems with respect to overhead and accuracy, in this section I would like to discuss how well the Energy Scheduling Class can be used in conjunction with normal interactive user programs.

5.6.1 Using Interactive Programs

As a first property of the Energy Scheduling Class, I wanted to identify how well signals, timers, and user input in programs cooperate with my energy measuring approach. For this purpose, I designed a micro benchmark, which measures how accurate the *sleep* system call works for a normal user program. Using this system call, a program can indicate to the kernel that it wants to wait for a specified amount of time. The Linux kernel implements this functionality by first removing the program from the scheduling process, then setting a timer at the specified time point, and last, as soon as the timer fires, adding the program back to the scheduling process. This implementation ensures that the program is not executed for the specified amount of time. However, if the system is loaded or if a higher priority program is currently executing, the sleeping program may need to wait longer than specified. A similar mechanism is also used for the delivery of signals and the management of user input. Consequently, the results of this micro benchmark indicate how well interactive programs can be used while there are energy measurements in parallel in the background.

To determine the influence of my energy measurement approach on different types of normal interactive programs, I executed the micro benchmark with different sleep lengths. For this purpose, I chose 200 ms, 500 ms, 1 s, and 2 s. A sleep duration of 200 ms represents a program with heavy interaction such as a text writing program, whereas a length of 2 s corresponds to a program with little interaction. The other two values I chose in the middle between

the two extreme values to be able to characterize programs with intermediate interactivity as well.

In parallel to the micro benchmark, I executed a program which creates background load. This allows me to characterize the sleep system call accuracy with different system load situations. The program creating the load uses for this purpose additional threads which execute a simple busy loop. I varied the number of threads from zero to four so that a system with no load at all as well as a heavily loaded system can be characterized.

Furthermore, I varied in which scheduling class the micro benchmark and the background load program were executed. The usual case and thereby my baseline is if both programs run in the Completely Fair Scheduling Class (CFS). By running the micro benchmark in CFS and the load program in the Energy Scheduling Class (EC), I was able to simulate a normal program with background energy measurements. With the reverse situation I could characterize a measured interactive program with other normal programs in the background. To determine the influence for a measured interactive program with other parallel background energy measurements, I executed the micro benchmark as well as the load program in the Energy Scheduling Class.

In Figure 5.7 the results of this experiment can be found. The four subfigures contain the results of the measurements with the four different sleep lengths. The bars in the plots represent the actual duration, which the program was sleeping. I included error bars in the graphs as well, if the standard deviation for the corresponding measurement was above five percent. With a perfect implementation, the measured sleep duration should be equal to the targeted one even if the system is heavily loaded.

As it can be seen in the figure, in the baseline situation, where the benchmark as well as the background program are executed in the Completely Fair Scheduling Class, the measured sleep duration exactly matches the targeted one. Even if the system is heavily loaded, no measurable delay occurs. If the load program runs in the Energy Scheduling Class, this situation looks completely different. With an increasing number of background load threads, the actually measured sleep duration increases significantly. The same pattern can also be found if the micro benchmark as well as the load program are managed by the Energy Scheduling Class. Besides the 200 ms version, the situation, where the benchmark is executed in EC and the load program in CFS, is equal to the baseline case.

The reason why interactive programs suffer if they are used in combination with other programs managed by the Energy Scheduling Class can be explained with the time slice round robin scheduling scheme, which is use by the Energy Scheduling Class internally to determine which program to run on the processor. This scheduling scheme does neither honor interactivity of programs nor respect dynamic priorities like the approach used by CFS. As soon as there is another program in the system, for which energy is currently measured, the usability of the interactive program gets reduced.

The reason why a measured highly-interactive program is also not very well usable even if there are only normal programs in the system, can be found in the *schdeduling class time-sharing algorithm*. Although, the interactive program has a higher priority, it is not selected for execution by the Energy Scheduling Class, because the time slice for the Completely Fair Scheduling Class, which was previously calculated by the class time-sharing algorithm, is not yet depleted. Hence, the load program continuous running even though the higher priority interactive program is ready again. If the sleep duration is increased to 500 ms the situation is

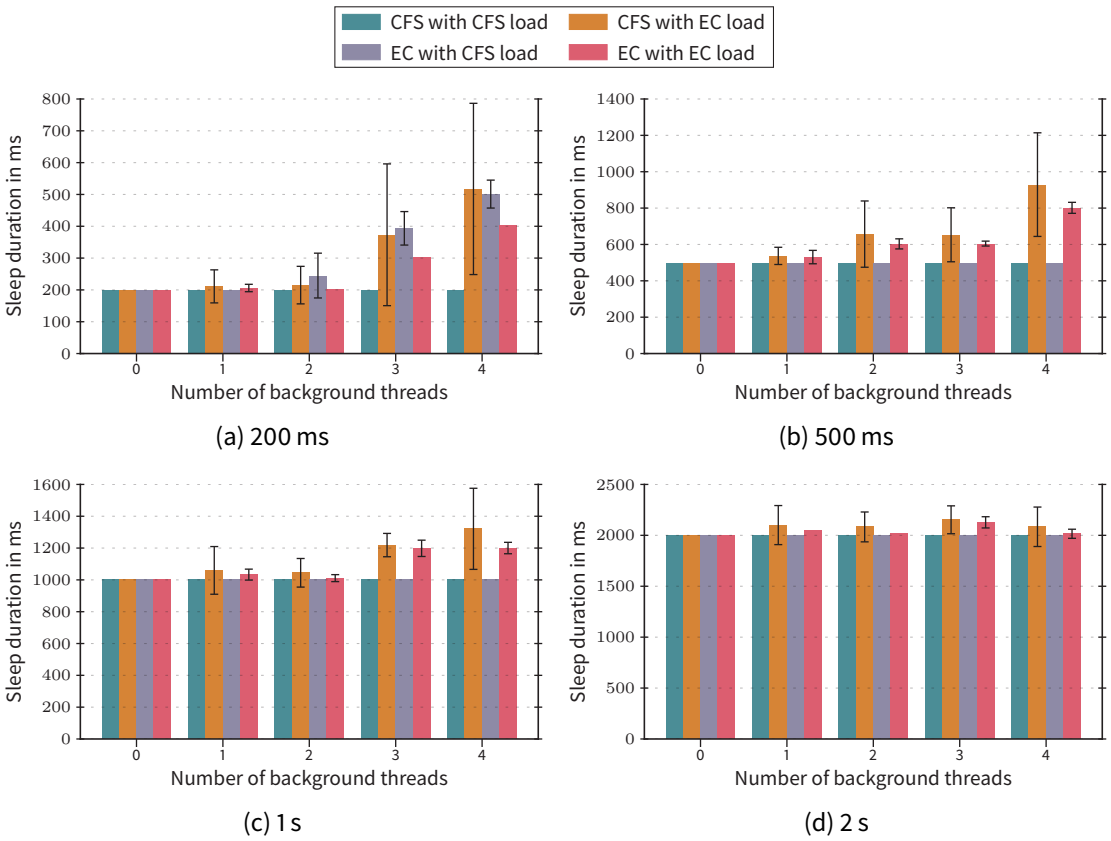


Figure 5.7: The results of the usability measurements of interactive programs. Each bar in the plots visualizes the time, which the benchmark actually slept with the different amount of threads producing background load.

different. The time slice for CFS is already depleted when the interactive program wakes up again, which in turn enables the Energy Scheduling Class to directly schedule the interactive program.

To put it in a nutshell, this experiment shows, that the time slice round robin scheduling scheme, which is use internally in the Energy Scheduling Class, is not well suited for interactive programs. Consequently, a future improvement of the Energy Scheduling Class should be to implement a better scheduling technique such as the one used by the Completely Fair Scheduler, which respects interactivity as well as program priorities.

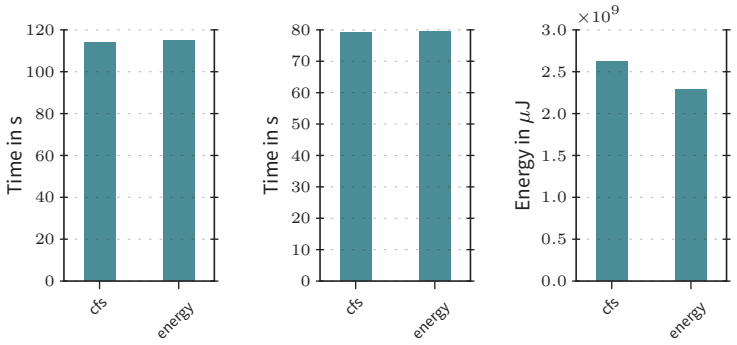
5.6.2 Measuring Interactive Programs

In addition to determining how much the Energy Scheduling Class influences the interactivity of normal user programs, I also wanted to know whether the Energy Scheduling Class can be used to collect energy statistics for interactive programs such as a text editor or browser. For this purpose, I designed another small micro benchmark, which simulates an interactive program. This benchmarks mimics interactivity by alternating between sleeping for some

time period and operating for some time period. How long the benchmark sleeps or works is decided based on a previously generated trace. Using a trace, allows me to have the exact same behavior every time when I execute the benchmark. The trace itself was created by generating 80 random values between 0 and 2000. One value in the trace represents the time in ms for which the benchmark has to either sleep or work. The generated trace, which I used for this experiment has an average work time of 1044 ms and an average sleep time of 870 ms. Hence, this trace represents a program with a rather intermediate interactivity like a web browser.

To have measurements, which I can compare against, I first executed the benchmark in the Completely Fair Scheduling Class in a stripped down Linux environment and measured the time the program was scheduled on the processor (CPU time), the overall runtime (wall-clock time), as well as the energy, which it consumed (package energy), similar to the baseline measurements presented in Section 5.2. Afterwards, I executed the benchmark in the Energy Scheduling Class and again measured CPU time and wall-clock time and additionally used the package energy as reported in the benchmark's *procfs* directory.

Since the trace corresponded to an intermediate interactive program and as there was no additional load in the system, the Energy Scheduling Class should have been capable of accurately measuring the energy consumption of the program with only a small runtime overhead.



(a) Wall-Clock Time (b) CPU Time (c) Package Energy

Figure 5.8: The results of the three measurement criteria of the measurements of interactive programs. Each bar in the plots visualizes the average of the 20 repetitions for the different parameters. Error bars are not shown in the graphs because the standard deviation of the results was below two percent.

As it can be seen in the results presented in Figure 5.8, no real difference in total runtime or CPU time is visible between the two measurements. The total runtime of the benchmark only increases by one second when executed in the Energy Scheduling Class, which corresponds to an overhead of less than one percent. Hence, the measuring overhead in this scenario is negligible. However, when looking at Figure 5.8c, it can be seen, that there is a significant difference in the reported energy consumption of the program. The reason for this difference is, that the baseline energy measurements, in contrast to ones done by the Energy Scheduling Class, are only correct if the measured program does not self-suspend but constantly executes. Since the benchmark simulates an interactive program, which self-suspends from time to

time to wait for user input, the baseline measurements not only contain the energy consumed by the benchmark but also the energy consumed by the idle task, which is executed if the benchmark sleeps. The energy measurement technique used by the Energy Scheduling Class, on the other hand, correctly handles the self-suspending of the program and accordingly only measures the energy when the program actually executes.

In total, this experiment shows that the Energy Scheduling Class can also be used to measure the energy consumption of normal interactive programs with nearly no runtime overhead. Furthermore, the experiment visualizes nicely why using a naive measuring approach is not a feasible solution to measure arbitrary programs. Even if nothing else runs in the system, the naive energy measurements already report incorrect values if the program uses self suspension. Only a measuring technique like the one used in the Energy Scheduling Class is capable to correctly account and measure energy in such a scenario, since it only measures energy when the program actually executes.

6 Conclusion and Future Work

6.1 Conclusion

Within this thesis I designed and implemented a new method to collect live energy measurements of individual programs in a multi-core system. The method is based on the energy consumption estimations provided by the RAPL energy counters available with recent Intel® processors. Since these counters differentiate neither between individual processor cores nor separate programs, I had to develop a special scheduling algorithm, which enables me to properly account and measure energy with the help of the counters in a multi-core system.

The scheduling algorithm, which I named *Locked-Out Gang Scheduling*, ensures that if a program, for which energy is currently measured, is executed on the processor, nothing else can run in parallel. Only the program with all its threads is allowed to execute. With this scheduling technique, the energy consumptions as reported by the RAPL counters can be accounted to the individually measured programs correctly.

Based on this algorithm, I implemented a new scheduling class in the Linux kernel, the *Energy Scheduling Class*, which allows live energy measurements of any program in the system. The scheduling class not only assures that only threads belonging to the same measured program can run simultaneously, but also guarantees that, when switches to other programs in the system occur, the consumed energy is properly accounted. An additional property of the Energy Scheduling Class is, that it integrates well into the existing scheduler hierarchy in the Linux kernel and thereby does not influence the system significantly.

During the evaluation I could show that this new energy measurement technique can provide accurate and low-overhead live measurements for arbitrary programs. For example, it was possible to determine the energy consumption of various benchmarks from the *NAS Parallel Benchmark* suite with less than six percent difference in energy and runtime. This high accuracy and low overhead could be achieved not only when the benchmarks executed solely in the system but also when they run together with some background load and even when two benchmarks were measured in parallel. Furthermore, with a self-designed micro-benchmark mimicking an interactive program, I could show that even for self-suspending programs the energy consumption can be determined with nearly no runtime overhead by my new energy measuring technique, which is not possible with any other measurement method available.

6.2 Future Work

Still, the presented design and implementation can be further improved. A significant limitation of the implementation of the scheduling class is for example, that it does not support multi-socket systems currently, which have separate RAPL counters on each socket. In such a

system it would be possible to run multiple programs in parallel on different sockets and still be able to correctly measure and account energy for them. Furthermore, a proper integration of higher priority scheduling classes such as the Deadline Scheduling Class is necessary for using the Energy Scheduling Class in conjunction with all other scheduling classes in the Linux scheduler without any restrictions. The current implementation does not handle higher priority scheduling tasks correctly. In addition to that, a better internal scheduling mechanism than the simple time slice round robin scheduling scheme should be used, which would allow a better support for interactive programs. As shown during the evaluation, highly interactive programs are hindered significantly if there are other measured programs in the system. Another improvement would be to replace the globally shared management data structures with multiple local replicas. The fact, that every core-local scheduler has to take a global lock during its scheduling process may be a critical performance bottleneck on larger systems. In addition, support in the Energy Scheduling Class to measure not only single programs but groups of them would also be a valuable enhancement. For example, such a functionality could be used to determine the energy of functional groups in the system or multiple interacting programs together.

In parallel to the mentioned improvements in the implementation of the Energy Scheduling Class, one could also investigate other methods to determine the energy consumption of the system. For example, one could integrate the energy counters available on recent AMD® processors or even use the feedback of real physical measurements. Such an integration of additional energy measuring methods would allow the usage of the Energy Scheduling Class on additional hardware, which does not provide the Intel® RAPL energy counters.

Summary

All in all, the energy measurement technique presented within this thesis allows fine grained measurements of arbitrary programs with a high accuracy and low runtime overhead. I could show, that the measurements are very reliable and precise even if they are taken at various situations, which are problematic for accurate energy measurements. Furthermore, the presented technique is, to my knowledge, the first measurement approach capable of accurately measuring the energy consumption of interactive programs in a multi-core system. Accordingly, I consider my new energy measuring technique an improvement over other already available methods.

Acknowledgment

I would like to thank my two advisers, Michael and Marcus, for their countless useful ideas, lots of valuable advises, and all the helpful criticism during the whole process of this thesis. Additionally, I want to thank Professor Hermann Härtig for giving me the chance to write this thesis and work with such a great team.

Bibliography

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[2] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 225–236. IEEE Computer Society, 2012.

[3] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX annual technical conference*, volume 14. Boston, MA, 2010.

[4] Liang-Bi Chen, Yen-Ling Chen, and Ing-Jer Huang. A Real-Time Power Analysis Platform for Power-Aware Embedded System Development. *J. Inf. Sci. Eng.*, 27(3):1165–1182, 2011.

[5] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors. `http://support.amd.com/TechDocs/49125_15h_Models_30h-3Fh_BKDG.pdf`, 2015.

[6] Jack Dongarra, Hatem Ltaief, Piotr Luszczek, and Vincent M Weaver. Energy footprint of advanced dense numerical linear algebra using tile algorithms on multicore architectures. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 274–281. IEEE, 2012.

[7] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[8] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5):658–671, 2010.

[9] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, Narayanan Vijaykrishnan, and Mahmut Kandemir. Using complete machine simulation for software power estimation: The softwatt approach. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 141–150. IEEE, 2002.

[10] Daniel Hackenberg, Thomas Ilsche, Robert Schone, Daniel Molka, Martin Schmidt, and Wolfgang E Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 194–204. IEEE, 2013.

[11] Daniel Hackenberg, Robert Schone, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the Intel Haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 896–904. IEEE, 2015.

[12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[13] Michael Hennecke, Wolfgang Frings, Willi Homberg, Anke Zitz, Michael Knobloch, and Hans Böttiger. Measuring power consumption on IBM Blue Gene/P. *Computer Science-Research and Development*, 27(4):329–336, 2012.

[14] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. SEEP: exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, page 4. ACM, 2011.

[15] Timo Hönig, Christopher Eibel, Wolfgang Schröder-Preikschat, Björn Cassens, and Rüdiger Kapitza. Proactive energy-aware system software design with SEEP. In *2nd Workshop EASED@ BUIS 2013*, page 9. Citeseer, 2013.

[16] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. Proactive energy-aware programming with PEEK. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, 2014.

[17] Ingo Molnar. [lkml] Modular Scheduler Core and Completely Fair Scheduler. `https://lwn.net/Articles/230501/`, 2007.

[18] Intel®. Intel 64 and IA-32 Architectures Software Developer's Manual. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`, 2015.

[19] Jake Edge. Perfcounters added to the mainline. `https://lwn.net/Articles/339361/`, 2009.

[20] Russ Joseph, David Brooks, and Margaret Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. In *Workshop on Complexity Effectice Design WCED, held in conjunction with ISCA*, volume 28. Citeseer, 2001.

[21] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[22] Shoaib Kamil, John Shalf, and Erich Strohmaier. Power efficiency in high performance computing. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[23] Z Nakutis. Embedded systems power consumption measurement methods overview. *MATAVIMAI*, 2(44):29–35, 2009.

[24] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `https://www.nas.nasa.gov/publications/npb.html`, 2016.

[25] OpenMP Architecture Review Board. Open Multi-Processing API Specification for Parallel Programming. `http://openmp.org`.

[26] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[27] Abhinav Pathak, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM, 2011.

[28] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the Intel® microarchitecture code-named sandy bridge. *IEEE Micro*, pages 20–27, 2012.

[29] Sebastian Ryffel. LEA2P-The Linux Energy Attribution and Accounting Platform. *Diploma thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland*, 2009.

[30] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 168–178. ACM, 2009.

[31] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[32] David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. Koala: A platform for OS-level power management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302. ACM, 2009.

[33] David C Snowdon, Stefan M Petters, and Gernot Heiser. Accurate on-line prediction of processor and memoryenergy usage under voltage scaling. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 84–93. ACM, 2007.

[34] TK Tan, A Raghunathan, and NK Jha. Emsim: An energy simulation framework for an embedded operating system. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 2, pages II–464. IEEE, 2002.

[35] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268. IEEE, 2012.

[36] Andreas Weissel and Frank Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246. ACM, 2002.

[37] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM, 2010.