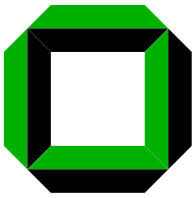

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Design and Analysis of Energy-Aware Scheduling Policies

Christoph Klee

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inform. Andreas Merkel

20. August 2008

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

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

Karlsruhe, den 20. August 2008

_____

Christoph Klee

Abstract

Scheduling policies of multitasking operating systems partition the time of processor assignment among all runnable threads. For most schedulers it is unimportant in which way a thread utilizes its assigned resources during its time of processor control. A thread's resource utilization, however, cannot only affect the thread itself, but also subsequent scheduled threads. In the case of a thread's power consumption, the power consumption can cause throttling of the thread and subsequent threads. Due to the interference, the processor allocation between the runnable threads is not fair any longer.

In this thesis, we propose a generic design to enhance general purpose schedulers to become energy-aware. The enhanced schedulers fairly partition the system's energy among threads to favor energy-efficient threads. Furthermore, they assure that a thread's power consumption does not affect another thread's execution negatively by enforcing another thread's throttling. In order to make best use of a processor's power limit, we present an energy transfer mechanism to fairly transfer energy among threads. It permits threads having power consumptions beyond a pre-defined power limit to benefit from threads having a power consumption below the limit.

Our evaluation shows that each of our examined schedulers can become energy-aware, and that they assure that each thread preserves the power limit individually. Besides, our enhanced schedulers permit to partition a system's energy fairly, even in the case of energy transfers. Due to the energy transfers and the fair energy partitioning, our enhanced schedulers – limiting each thread's power consumption – achieve a better performance than schedulers only limiting a run-queue's and a processor's power consumption, respectively.

Contents

Chapter 1

Introduction

In this chapter, we motivate why energy-aware schedulers preserving a power limit and favoring energy-efficient applications can be an advantage for data centers. We analyze which requirements energy-aware schedulers must fulfill to permit data centers to bill an application's caused power consumption. Furthermore, we analyze how these schedulers can allow applications having power consumptions beyond the power limit to benefit from applications having power consumptions below the power limit. Subsequently, we present our solution for these two problems, and outline the structure of our thesis.

1.1 Problem Analysis & Solving

Motivation Today's general purpose operating systems are multitasking capable. They do not grant a thread – the schedulable entity of an application – processor control until it finishes its execution, instead they execute the thread only for a short period of time. This period is called *timeslice*. By preempting a thread after it has executed for the duration of its timeslice and scheduling another thread, it seems as if a scheduler would execute several threads in parallel on one physical central processing unit (CPU).

A scheduler of a multitasking operating system (OS) is responsible for scheduling threads. Depending on a thread's characteristics, as e.g., priority or its course of execution, a scheduler can determine a thread's timeslice length; it can even assign a pre-defined timeslice length to all threads. After a thread has executed for the duration of its timeslice, it will at the latest get preempted if more than one thread is eligible to be scheduled. If a scheduler is priority based, a higher priority thread can preempt the currently running thread as well.

For a scheduler of a general purpose OS it is mostly unimportant in which way a thread utilizes its assigned resources during its timeslice. This will be a drawback if an uncontrolled resource utilization reduces the system's performance. The power dissipation, in particular, of a processor caused during a thread's execution can raise a processor's temperature above its critical threshold. If this happens, a processor must be throttled to reduce its power dissipation and temperature [31].

Due to the thermal resistance of a processor's heat sink, a processor's temperature changes slowly [14]. Hence, a processor's throttling to reduce its temperature can last for several timeslices and therefore can affect subsequently executed threads [10].

Thus, a thread's execution can have negative impacts on the execution of subsequent threads. The throttling policy implemented in hardware of a processor or an energy-aware scheduler can throttle the processor. A scheduler can control which thread it can throttle and for how long. Furthermore, it can even throttle threads before they raise the temperature of the processor above a threshold.

A processor cannot only be throttled to avoid an exceedance of a processor's temperature above a threshold, additionally it can be throttled to restrict its average power consumption. The reasons for restricting a processor's average power consumption can be multifaceted, e.g., a restriction of a power supply, a critical temperature of a processor as motivated before or even monetary reasons.

A monetary reason can be the costs for a computer room air conditioning (CRAC) system [28]. Normally, a CRAC must at least have a cooling capacity to discharge the hot air caused by the components of a data center with cool air, even if these components consume their maximum power. If the components may not consume more power than permitted by a power limit, the CRAC can be sized smaller to reduce the CRAC's acquisition and operating costs.

Although the cooling costs of a CRAC and the energy costs to operate servers depend on the power dissipation of the executed applications in a data center, it is more usual to pay for an application's execution time, but not for its caused energy costs. Sun offers, in particular, its "Sun Grid" [36] computing power for one dollar per CPU-hour. An application will execute for one CPU-hour if it uses k nodes for its $\frac{1}{k}$ hours lasting execution. Similar is Amazon's approach to sell the compute power of their "Amazon Elastic Compute Cloud" [2, 23]. The price for the compute power depends on the requested resources as data storage, memory and number of compute units. Amazon considers how much resources a customer requires and bills the performed data transfer, but does not consider the energy costs resulting from the utilization of the assigned resources.

Analysis For permitting a company to sell its compute power and to bill the caused energy consumption, a scheduler must preclude drawbacks from other threads' power consumptions. To avoid these drawbacks and unfairness, respectively, which are caused by a thread's power consumption, an energy-aware scheduler must consider to limit a processor's power consumption by throttling the threads individually [6, 47] and not the complete run-queue.

In addition to limiting a thread's power consumption, an energy-aware scheduler can – independently of a power limit – base its scheduling decisions on the energy consumption of the system's threads. If a system's energy is supposed to be fairly partitioned among the threads eligible to be scheduled to favor energy-efficient threads, threads having the same characteristics – as previously outlined – must get the same amount of energy. This means, a scheduler will no longer execute a thread for the duration of its timeslice, but as long as its assigned energy will last. Thus, a scheduler assigns a thread with a lower power consumption to the processor for a longer time than a thread with a higher power consumption.

Due to the threads having power consumptions below the power limit, a processor's average power consumption can be below the permitted power limit. This permits threads with power consumptions above the limit to exceed the power limit as long as the processor's average power consumption is not raised above the limit. This energy transfer must fairly partition the transferred energy to avoid unfairness among the receiving threads. Besides, a scheduler must limit the accumulation of energy which it

can transfer, otherwise – if threads consume the energy within a short period of time – threads can exceed the permitted energy consumption or processor temperature.

Solution In this thesis, we propose a generic design to enhance general purpose schedulers to become energy-aware. The enhanced schedulers can fairly partition the system's energy among threads and can assure that a thread's power consumption does not negatively affect another thread's execution by enforcing another thread's throttling. For limiting a thread's power consumption, our energy-aware schedulers throttle a thread to assure that a thread does not raise its average power consumption above a limit. This does, however, not guarantee a fair energy partitioning. Therefore, a schedulers only execute a thread until the thread has consumed the energy permitted by an energy limit. Thus, a thread's scheduling period does no longer depend on its timeslice length, instead it depends on the assigned energy a thread may consume. After a thread has consumed its assigned energy, a scheduler schedules the next thread. If it is not desired to execute a thread for a longer period than its timeslice, a scheduler can discard a thread's remaining energy and can schedule another thread.

In addition to preserving the power limit per thread, our energy-aware schedulers can permit threads having power consumptions beyond the power limit to benefit from threads having power consumptions below the limit. Therefore, we have designed an energy transfer between threads. As pointed out, the transferred energy must be fairly partitioned, hence a first come first serve scheme is insufficient for energy transfers. We partition the offered energy, before a scheduler transfers the energy to threads having power consumptions above the power limit.

The power limit and the energy limit are independent of each other. We have outlined that each limit can be accomplished in two different fashions. An energy-aware scheduler can prohibit or permit threads to benefit from another one's power consumption, and it can schedule a thread at most as long as the thread's timeslice last or until the thread has exhausted its assigned energy. Thus, we investigate four different energy policies.

1.2 Structure

Our thesis is structured as follows: In Chapter 2, we present the scheduling policies we analyze. Furthermore, we point out how the kernel can reveal a thread's energy and power consumption as well as a processor's temperature from a processor's performance counters and how the kernel can limit them. Chapter 3 deals with our generic design to enhance schedulers to become energy-aware and to move away from a timeslice based scheduler to an energy based scheduler. These schedulers permit to partition a system's energy fairly among threads. We consider the implementation of our design on top of the Linux kernel in Chapter 4. In addition to implementation details applying to all of our analyzed schedulers, we examine scheduler specific adaptions to allow for scheduler specific characteristics. Afterwards, we evaluate the studied schedulers and the realization of the fair partitioning of the system's energy in Chapter 5. In Chapter 6, we present related work regarding the fair energy partitioning. At the end of our thesis, we point out our achievements, give a short summary of our work and outline possible directions of future work in the last Chapter 7.

Chapter 2

Background

We define in this chapter terms required for our thesis. Afterwards, we present the seven scheduling policies which we have examined and adapted to become energy-aware. Then we introduce mechanisms to account a thread's energy and power consumption as well as a mechanism to estimate a processor's temperature. At the end, we outline a mechanism to limit a processor's power consumption.

2.1 Terminology

In this thesis, we will use the following terms to define the subsequent abstractions:

Thread A thread is a schedulable entity, reflecting a single flow of execution. Each thread stores its own program counter – to state which instruction is executed next – as well as registers, stack and status. If a thread is executed, its register state is stored in the physical registers of a processor, otherwise it is stored in memory.

Task A task is composed of its assigned threads running within one address space. At least one thread must be assigned to a task.

Timeslice A timeslice is the period of time a scheduler intends to schedule a thread for. It is assigned to a thread.

Quantum If a scheduler schedules a thread until it has consumed its assigned energy, a thread can exhaust its assigned energy before or after its timeslice expires. Therefore, we must distinguish between the period of time a scheduler designates to schedule a thread for (timeslice) and a thread's actual execution time. We call the actual execution time quantum.

Run-Queue A run-queue is a queue containing runnable threads of a processor.

2.2 Scheduling Policies

We present in this section the scheduling policies we have investigated and adapted to become energy-aware. At first, we introduce the round robin and multilevel feedback

queue policies. Afterwards, we consider three proportional share scheduling policies and, at last, we examine the two schedulers of the Linux 2.6 kernel series.

2.2.1 Round Robin

The round robin scheduling policy [37] is a preemptive timeslice based scheduling algorithm. It picks the first thread of a processor's run-queue, then executes the thread until its timeslice expires, or the thread blocks or yields processor control back to the kernel. In the case of a blocking thread, the scheduler removes the thread from the run-queue, otherwise it reinserts the thread at the end of the run-queue. This assures that after n schedules each thread of the run-queue is scheduled for one timeslice, while n is the number of threads of the run-queue. Round robin treats threads equally, so that it can assign a default timeslice length to each thread. Hence, round robin is a starvation-free scheduling policy. Besides, round robin is an $O(1)$ scheduler, because its scheduling complexity is independent of the number of threads assigned to a run-queue.

Due to the fact that round robin treats threads equally, it inherently penalizes input/output-bound (I/O-bound) threads and favors CPU-bound [46] threads. An I/O-bound thread normally exhausts only a fraction of its timeslice in contrast to a CPU-bound thread. This effect can be reduced by a shorter timeslice length, because then I/O-bound threads get the chance to be executed more frequently. The disadvantage of a shorter timeslice length is the increased address space switching overhead.

2.2.2 Multilevel Feedback Queue

A multilevel feedback queue (MLFQ) scheduler [39] is a preemptive priority based scheduling algorithm. Each thread has an assigned priority. These priorities define an order among the threads. If a thread becomes runnable and has a higher priority than the currently running thread, it will preempt the current thread. This means the current thread will no longer be permitted to execute while the higher priority thread will be permitted to execute. Thus, a priority based scheduler assures that it executes one of the runnable threads having the highest priority among the runnable threads.

A MLFQ scheduler consists of multiple queues, where each queue is associated with a priority and a timeslice length. Its first queue, linked with the highest priority, has the shortest timeslice length. The more a queue's priority decreases, the more its timeslice length is extended. A MLFQ scheduler executes the threads assigned to a queue in round robin fashion.

If a thread becomes runnable for the first time, the scheduler will insert it at the end of the first queue assigned to the highest priority. As long as the scheduler has only executed a lower priority thread before or the processor has been idle, the created thread preempts this thread and starts with its execution, otherwise it waits to be selected by the scheduler. The preempted thread stays at the front of its queue to get the chance to execute for the remaining time of its timeslice.

In the case a thread executes only for a fraction of its timeslice because it blocks, the scheduler will increase its priority, otherwise the scheduler will decrease it. Thus, a MLFQ scheduler ensures that I/O-bound threads execute with a high priority and preempt low priority CPU-bound threads. Furthermore, the length of a low priority thread's timeslice accommodates the requirements of a CPU-bound thread.

The MLFQ scheduler is a priority based scheduling algorithm, therefore it suffers from starvation. If high priority threads can utilize the processor to 100%, a low priority

thread will suffer from starvation, because the scheduler will not execute the thread as long as higher priority threads are runnable. Torrey, Coleman and Miller [39] have not considered starvation in the design and implementation of their MLFQ scheduler. To overcome this drawback, we introduce an aging mechanism. Each thread's age is equal to the timeslice length corresponding to the thread's priority. The scheduler assigns the age to a thread whenever it refreshes a thread's timeslice.

To avoid starvation, the scheduler must at least age the threads which have the lowest priority among a processor's threads after k executions of higher priority threads if high and low priority threads are runnable. The MLFQ scheduler may raise an aged thread's priority up to the highest priority if it does not execute the thread at a lower priority before.

Our aging mechanism works as follows: it decreases the age of all lower priority threads about the timeslice length of the currently executed thread. If a lower priority thread's age is afterwards equal or less than zero, the scheduler will raise its priority. Thus, an $O(n)$ aging mechanism can avoid starvation of lower priority threads.

2.2.3 Start-time Fair Queuing

Start-time fair queuing (SFQ) is a proportional share scheduling algorithm [16]. Before we present the SFQ scheduler, we outline the general concept of proportional share scheduling policies.

Proportional Share Policy A proportional share scheduling policy strives to partition the time of processor allocation fairly among the threads according to the weights of the threads. Each thread has an assigned weight, the threads' weights define the relation among the threads. A thread with k times the weight of another thread, executes k times longer than the other thread in a period of time. Furthermore, a thread with weight r shall get the portion $\frac{r}{r_{\text{total threads}}}$ of the processor allocation, while $r_{\text{total threads}}$ is the sum of all threads' weights of a processor. The processor allocation will be fair for a processor's runnable threads if each thread gets exactly its portion of time within each time interval $[t_i, t_j]$. This is only an idealized definition of a proportional share scheduling policy, because a thread cannot allocate a processor in arbitrarily small units.

Therefore, most proportional share scheduling policies have a concept of *virtual time*. The virtual time defines which thread is scheduled next. Proportional share schedulers based on virtual time schedule on of the threads with the smallest virtual time among all threads of a run-queue. Each thread has its own virtual time, which is synchronized with a per run-queue global virtual time when a thread becomes runnable. The idealized idea of virtual time is that the virtual times of all threads and the global virtual time of a run-queue are equal at each distinct point of time.

Only during a thread's execution its virtual time increases. The greater a thread's weight is, the slower its virtual time increases. The different increase of the threads' virtual times per physical time unit assure a proportional share among the threads according to their weights. As mentioned before, a thread's virtual time is synchronized with a per run-queue global virtual time when a thread becomes runnable. The greater the total weight of all threads of a run-queue is, the slower the global virtual time increases. Proportional share schedulers measure the virtual time in time units. A time unit can be, e.g., a timer tick, a processor cycle or even a physical time unit such as a nanosecond.

SFQ Scheduler SFQ strives to minimize the unfair processor allocation caused by the coarse-grained processor allocation. This is achieved by introducing a virtual time v and extending each thread's thread control block with a start tag S and a finish tag F. The start tag corresponds to a thread's virtual time when the scheduler inserts the thread into the run-queue. This happens when a thread becomes runnable or after it has lost processor control and has to compete again for processor control. A thread's finish tag denotes a thread's virtual time after its last execution. After a thread's execution, it is equal to a thread's start tag S increased by the weighted period of time of the thread's last quantum it has executed for. A thread's quantum lasts at most as long as the thread's timeslice. To realize a proportional share among threads, SFQ schedules a thread with k times the weight of another thread, k times more often than the other thread.

The start tags define the scheduling order of the runnable threads. If several threads have the same start tag, the scheduler will break the tie arbitrarily. When the scheduling algorithm is initialized, the virtual time and each thread's start and finish tag are set to zero. As long as the processor is not idle, the scheduler schedules one of the threads with the smallest start tag and virtual time, respectively. Otherwise, the scheduler sets its virtual time to the maximum of all finish tags. The global virtual time will only increase if the start tags of all threads are greater than the global virtual time or the processor is idle.

As outlined before, a thread's finish tag is incremented only by the weighted quantum a thread has executed for and not by one weighted timeslice. Therefore, a scheduler may schedule blocking threads more frequently than non-blocking threads with the same weight due to the blocking threads' smaller start tags and virtual times, respectively. This guarantees a fair proportional share between CPU-bound and I/O-bound threads. Thus, SFQ is a starvation-free scheduling policy.

2.2.4 Lottery Scheduling

Lottery scheduling is a randomized proportional share scheduling algorithm [42]. In contrast to the remaining proportional share scheduling policies, it does not have the concept of virtual time. It selects a thread to be scheduled for the period of one timeslice by holding a lottery. Each runnable thread holds tickets according to its weight. This permits to realize a proportional share among threads. The scheduler allocates the tickets in the order of the threads in the run-queue. The first thread holding j_1 tickets, holds the tickets 1 up to j_1, the second thread holding j_2 tickets, holds the tickets j_1+1 up to j_1+j_2 and so on. The randomized scheduling algorithm chooses a ticket number m between one and the total number of tickets held by all runnable threads. If the first thread of the run-queue holds $\geq m$ tickets, then the first thread will be scheduled. Otherwise, if the first thread holds only $j_1 < m$ tickets, then the scheduler will check which one of the following threads holds the winning ticket. Thereto, the scheduler selects the k^{th} thread of the run-queue fulfilling the following two inequations:

$$j_1 + ... + j_{k-1} < m \tag{2.1}$$

$$j_1 + ... + j_k \geq m \tag{2.2}$$

In contrast to SFQ, lottery scheduling is solely probabilistically fair. This means, it is not possible to bound the unfairness between two threads for a given interval, because "the actual allocated proportions are not guaranteed to match the expected proportions

exactly" [42]. Nevertheless, lottery scheduling does not suffer from starvation, because each thread holds at least one ticket t, its probability to win a lottery is $p = \frac{t}{T}$, while T is the total number of tickets of all runnable threads.

In the case a thread blocks before it has consumed its complete timeslice, a fair share between this thread and another one exhausting its complete timeslice cannot be guaranteed any longer. Therefore, if a thread has consumed only a fraction f of its timeslice, the scheduler will scale the thread's tickets with $\frac{1}{f}$ for the next time the thread becomes runnable. Consequently, when the thread becomes the next time runnable it has a $\frac{1}{f}$ times greater chance to be elected by the scheduler.

2.2.5 Stride Scheduling

Stride scheduling is a proportional share scheduling algorithm [41]. It is based on the concept of virtual time outlined in Subsection 2.2.3. The basic idea of the stride scheduling policy is to represent the time interval a thread has to wait between its consecutive executions. This time interval is called *stride*. A thread's stride is inversely proportional to a thread's assigned number of tickets. Thus, a thread with twice as many tickets as another thread has to wait half as long as the other one to get executed, because its stride is only half as long as the other thread's stride.

A thread's stride is measured in virtual time units called passes. Each thread has its own virtual time. After a thread has executed for its timeslice, the scheduler updates a thread's virtual time. It increases a thread's virtual time by a thread's stride. In comparison to the SFQ's start tag, the virtual times of all runnable threads define the scheduling order. Hence, the stride scheduler schedules the thread with the smallest virtual time. If several threads have the same virtual time, the scheduler will break the tie arbitrarily. In order to account for threads not having used up their complete timeslices, the scheduler merely increments a thread's virtual time by the scaled stride. The scheduler scales a stride by the fraction of the timeslice which the thread has executed for. Like the two previous proportional share schedulers, the stride scheduler schedules a thread with k times the number of tickets of another thread, k times more often than the other thread.

In addition to a thread's virtual time, stride and number of tickets, the scheduler maintains a global virtual time, a global stride and a global number of tickets per run-queue. A run-queue's global stride is inversely proportional to the run-queue's global number of tickets. The scheduler requires the global virtual time to synchronize a thread's virtual time with the global one, whenever it enqueues a thread into the run-queue. Otherwise, a thread blocking for a long period of time may monopolize the processor until its virtual time is larger than any other thread's virtual time of the run-queue. In order to avoid that the scheduler schedules each enqueued thread at next, the scheduler increments a thread's synchronized virtual time – equal to the global virtual time – with the remaining number of passes which have to pass until the thread's next execution. If the scheduler enqueues a thread for the first time into the run-queue, a thread's remaining passes will be equal to its stride. Otherwise, they will be equal to the passes which had to pass until the thread's next selection when the thread was removed from the run-queue.

In analogy to the dependency of a thread's virtual time on a thread's tickets or stride, respectively, the global virtual time depends on the global number of tickets hold by all runnable threads of a run-queue. Therefore, the scheduler must update the global number of tickets and the global stride if a thread joins or leaves a run-queue as well as if a user changes a thread's number of tickets while it is runnable. Similar to a

thread's virtual time, the scheduler increments a global virtual time by the global stride scaled by the elapsed time having passed since the last update of the global virtual time. As the other introduced proportional share scheduling algorithms, stride scheduling is starvation-free.

2.2.6 $O(1)$ Linux Scheduler

The Linux $O(1)$ scheduler was introduced with the Linux 2.6 kernel series. It is an $O(1)$ priority based scheduling algorithm favoring I/O-bound threads. The scheduler reserves the first hundred priorities for real time threads. It can assign conventional threads to the remaining forty priorities 100 (highest priority) to 139 (lowest priority) [39]. In contrast to the introduced MLFQ scheduler (cf. Section 2.2.2), a thread with priority 100 has the longest lasting timeslice and a thread with priority 139 the shortest. Each priority has its own priority run-queue, to allow the scheduler to schedule the threads of a priority in round robin fashion.

As long as at least one thread is runnable, the $O(1)$ scheduler schedules the thread at the head of the first non-empty priority run-queue. This thread belongs to the group of runnable threads having the highest priority among the runnable threads. Otherwise, the scheduler schedules the idle thread. In order to favor I/O-bound threads, the scheduler bases a thread's priority on a thread's static priority set by the user as well as on a thread's dynamic priority bonus determined by the scheduler itself. Due to the dynamic priority bonus, a thread's priority can diverge from its static priority about ± 5. I/O-bound threads can get a dynamic priority bonus of up to -5 and CPU-bound threads can get a penalty of up to $+5$, labeled as interactive and batch threads, respectively. The scheduler bases a thread's dynamic priority on a thread's runtime and the time a thread sleeps waiting for becoming unblocked. A thread's dynamic priority credits a thread for its sleeping time and penalizes it for its execution. The timeslice length of a thread, however, depends solely on a thread's static priority and not additionally on its dynamic priority bonus.

A run-queue is organized as a priority queue, hence lower priority threads may suffer from starvation. To prevent starvation and to guarantee the $O(1)$ behavior of the scheduler, a run-queue consists of two priority arrays: an active and an expired priority array. After a thread of the active array has executed, the scheduler inserts it into the expired array. The scheduler can at the earliest schedule the thread after the active array is empty. When the active array is empty, the scheduler switches the arrays. Thus, the old active array is the new empty expired array and the old expired array the new active array containing all runnable threads.

A drawback of this approach is that an interactive high priority thread may wait for a long period of time to be executed. Therefore, if a thread has exhausted its timeslice, the scheduler will check whether the thread is interactive or not. If it is not an interactive thread, the scheduler will insert the thread into the expired array, otherwise it will usually reinsert the thread at the tail of the priority queue of the active array. This special treatment of interactive threads will cause starvation of expired threads if interactive threads can completely utilize the processor. Thus, the scheduler must insert an interactive thread into the expired array under the following conditions:

1. Its priority is less than the highest priority of a thread in the expired array.
2. The first expired thread has waited already for a sufficiently long time, which depends on the number of runnable threads in a processor's run-queue.

At last, the scheduler improves the interactivity between high priority interactive threads by forbidding an interactive thread to exhaust its timeslice in one piece, instead its timeslice is split into several pieces ensuring a better reactivity of threads having the same priority. Otherwise, a thread would monopolize the processor for a long period, as in the case of high priority batch threads due to the length of their high priority timeslices.

2.2.7 Completely Fair Scheduler

The completely fair scheduler (CFS) is a proportional share scheduling algorithm introduced with the Linux kernel 2.6.23. Its aim is to model an "ideal, precise multitasking CPU" [9] on real hardware. Such an ideal CPU does not exist, therefore CFS executes the thread with the greatest demand for processor time among the runnable threads. This assures that all threads get an equal share of processor time proportional to their weight. As SFQ and stride schedulers, CFS is based on the concept of virtual time (cf. Subsection 2.2.3).

In contrast to the previous $O(1)$ Linux scheduler, the CFS is no longer based on run-queues and priority arrays. Instead, a red-black tree models the future scheduling order of the runnable threads. Consequently, CFS is not an $O(1)$ scheduler, but an $O(log(n))$ scheduler.

CFS schedules the thread with the least difference between the thread's virtual time and the per-processor virtual time. A per processor virtual time increases by the time having passed since its last update scaled with the total weight of all threads. The greater the total weight of all threads of a processor is, the slower the per processor virtual time increases. A thread's virtual time only increases by its weighted execution time during a thread's schedule. The greater a thread's weight is, the slower increases the thread's virtual time.

Contrary to the per priority fixed timeslice length of the old $O(1)$ scheduler of Linux, CFS has dynamically determined timeslices of variable length due to the following two reasons:

1. If the currently executing thread's virtual time is greater than the smallest virtual time in the red-black tree, while the scheduler allows a small gap between both virtual times to avoid an over-scheduling of threads and trashing of a processor's cache, the thread with the smallest virtual time will preempt the current thread.

2. In order to improve the reactivity of interactive threads, the scheduler can ignore a thread's sleeping time. This will be the case, if a thread wakes up from sleeping and has slept for less than a certain period defined by a tunable scheduling latency. Consequently, it is possible to place a thread at the front of the scheduling timeline which an event has just woken up.

Unlike waking up a sleeping thread and possibly inserting it at the front of the timeline, the scheduler inserts a newly woken up thread at the end of the timeline. The same applies to threads yielding processor control back to the kernel.

In contrast to CFS, the previous proportional share scheduling algorithms have realized the proportional share by scheduling threads more frequently according to the threads' weights. CFS schedules a thread with k times the number of tickets of another thread, k times longer than the other thread, but not more often. The other difference between CFS and the previous proportional share schedulers is that the previous proportional share schedules define a proportional share only between all threads of a

run-queue. CFS defines an additional hierarchy between these threads. This hierarchy groups threads together either based on their user ids or on their assignment to an administrative defined group of threads (e.g., a group of browsers). Thus, the weight of a thread does only influence its proportional share within its assigned group. The group weight defines the proportional share between the different groups.

CFS is a proportional share scheduling policy and each thread as well as each group gets a non-zero fraction of the system's share, hence it is starvation-free and therefore the CFS – unlike the $O(1)$ scheduler – does not need to check whether a thread starves.

2.3 Energy Accounting

With the rising energy costs, energy becomes a first class operating system resource [47]. To be able to manage energy as a resource, a scheduler requires to account the energy consumption caused by a thread's activities. Due to the increasing complexity of modern microprocessor architectures, a processor's energy consumption cannot be revealed from its utilization and duty cycles, respectively, any longer, because a processor's power consumption shows a wide variety for a given processor utilization [45]. For older microprocessors as the Pentium 2 processor, this was possible due to the simpler architecture [32].

Nevertheless, for limiting a processor's power consumption or preventing an exceedance of its temperature above a threshold, it is necessary to account a processor's energy consumption or at least its temperature. Bellosa et al. propose to correlate a few processor's events with their caused energy consumptions [6] to estimate a processor's energy consumption. The energy consumption can be estimated if the events cover a processor activities contributing mostly to a processor's energy consumption. To account the relevant events, they use a processor's performance counters. Most modern processors have performance counters; they are responsible for accounting specific events as mispredicted branches or cache misses [1, 18]. By accounting and weighting the relevant events, a processor's energy consumption can be estimated. Our proposed energy policies require this mechanism in order to decide how long they may schedule a thread if the scheduling decision is based on a thread's estimated energy consumption.

Power Consumption To limit a thread's power consumption it is necessary to know its current power consumption or at least its average power consumption caused during the last k milliseconds. The approach of Bellosa et al. to estimate a thread's energy consumption can also be used for estimating a thread's power consumption, because energy is the product of power and time. By periodically accounting a thread's energy consumption every k milliseconds, its average power consumption can be accounted.

A thread's power consumption can change from time to time because of distinct phases of execution of a thread. The power consumption can change more frequently if a scheduler throttles a thread for very short periods of time like the period of one timer tick. An exponential average can smooth the effect of accounting a quickly changing power consumption. If it is not possible or not intended to account a thread's energy consumption every k milliseconds, an exponential average can be enhanced to allow for sampling periods of variable length of a thread's energy consumption [25]. An *energy profile* contains a thread's exponential average energy consumption and its total consumed energy. To limit a thread's power consumption, our energy policies require the exponential average energy consumption. For a fair energy partitioning, we need the total consumed energy.

2.4 Temperature Estimation

A processor's consumed energy is dissipated as heat [33]. Its temperature and power consumption are closely coupled quantities [3], because a processor's power consumption increases its temperature and vice versa. To reduce a processor's temperature, a heat sink mainly accumulates a processor's dissipated energy; a part is stored within the processor itself. A heat sink can slow down an increase of a processor's temperature. Due to the limited capacity of a heat sink, it emits heat to the ambient air [20].

A heat sink can be modeled as a thermal capacitor being charged and discharged like an electrical capacitor [17]. Due to the exponential course of charging and discharging a capacitor, a processor's temperature course is exponential as well.

To measure a processor's temperature, its thermal diode can be used, but reading the diode takes several milliseconds (about $10ms$ on a Pentium 4) [21]. To avoid this overhead, Bellosa et al. propose a thermal model for a heat sink. It permits to predict a processor's current temperature without reading a thermal diode. This model depends on the heat sink's characteristics, the ambient temperature and the processor's energy consumption [6, 20].

Skadron et al. [22] propose a more fine-grained solution. They use performance counters and a processor's floorplan to predict the temperature of individual functional units. For our work Bellosa's thermal model is sufficient.

2.5 Dynamic Thermal Management

With the increasing complexity of modern microprocessors and their applied on-chip power management techniques, the gap between a processor maximum power consumption and its typical power consumption increases even further [8]. To be able to deal with a processor's maximum power dissipation, a processor's cooling techniques must be designed for it. The total integration costs for processors having maximum power dissipations above 35-$40W$ increase with each Watt about $\$1$ [38], hence it will be an advantage if dynamic thermal management (DTM) techniques can reduce the maximum power dissipation.

In this case, the cooling techniques merely need to be capable to deal with the typical power dissipation of a processor, but not with the maximum power consumption. One or more DTM techniques are responsible for reducing the processor's power consumption and temperature, respectively. It depends on the DTM technique how long its response delay lasts, until it starts to reduce the processor's power consumption and what the resulting performance loss is.

Next, we discuss CPU throttling as a mechanism to limit a processor's power consumption. More DTM techniques such as voltage and frequency scaling as well as instruction cache throttling or speculation control [8] are out of scope of this thesis. Therefore, we do not discuss them.

CPU Throttling For reducing an idle processor's power consumption, several architectures have a special instruction. This special instruction permits the processor to switch to a low power state. Due to an external event, the processor switches again to the running state. Additionally, this instruction, e.g, the `hlt` instruction of x86 processors [11] gives the opportunity to throttle a thread by executing the instruction during the thread's execution [5]. Although throttling does not minimize a processor's max-

imum power consumption, it can reduce a processor's average power consumption as well as its temperature.

If the throttling results in quick changes of a processor's power consumption, the usable capacity of a battery can be significantly reduced in comparison to discharging the battery with a continuous load having the same average power consumption [24]. Therefore, the throttling mechanism has to avoid those quickly changing power consumptions if the system is battery based by executing the special instruction for several milliseconds.

Chapter 3

Design

Multitasking operating systems partition the time of the processor assignment among all runnable threads of a system. Thereby, it seems as if the threads execute in parallel, although they might only execute on one physical CPU. The operating system's scheduling policy decides which thread it schedules next and for how long it executes the thread. Its scheduling decision is based on a thread's properties as priority, weight or position in the processor's run-queue.

Most scheduling policies do not base their scheduling decision on utilization of resources assigned to a thread during its processor control and the consequences resulting from its utilization for other threads. A thread's energy consumption during its quantum, in particular, has usually no impact on a scheduling policy assuring a fair processor assignment. The reason for this is that a scheduler considers the fair processor assignment only in the time dimension, but not in the dimension of energy consumption.

In an environment where the energy consumption is limited over a period of time of a processor, due to limits of a power supply or thermal considerations, it will be an advantage if a scheduler fairly partitions the energy among all threads.

At first, we motivate why a fair energy partitioning will be an advantage if a processor's power dissipation is limited or its temperature must not be raised above a threshold. Afterwards, we outline that to guarantee a fair energy partitioning and to limit a processor's power consumption we require energy and power limits. Then we introduce four energy policies which strive to assure a fair energy partitioning. At last, we propose a design for a per run-queue energy budget permitting to transfer energy among threads, and we point out how a scheduler can preserve a thread's energy limit.

3.1 Problem Description & Analysis

Most scheduling policies only consider to partition the time of processor assignment among the processor's assigned threads. In the case of priority based scheduling policies, a scheduler fairly partitions the time between threads belonging to the same priority. Nevertheless, from the point of view of fair processor time partitioning, the partition cannot be fair if a thread's resource utilization has negative effects on the successive execution of other threads. We will consider a thread's execution as negative for other threads, if other threads cannot achieve at least the same during their quanta, as without running the thread additionally.

This is the case, in particular, in an environment with limited energy resources or a critical processor temperature which may threads not exceed. A thread's power consumption can force other threads to consume less power to prevent an exceedance of the processor's critical temperature or due to a limited amount of energy. This thesis addresses mechanisms to avoid an exceedance of processor's energy consumption over a period of time or of its critical temperature by limiting and fairly partitioning a processor's energy.

Power Consumption Limitation Limiting the energy consumption of a run-queue over a period of time, is equivalent to limiting its and the processor's average power consumption, respectively. Then, threads consuming more energy than assigned to them within their quanta – having average power consumptions beyond the allowed power limit –, force other threads in the run-queue to consume less energy than assigned to them during their quanta, to meet the overall per run-queue energy consumption limit.

Dynamic frequency or voltage scaling [8, 13, 15, 43, 44] as well as throttling [5, 45] can achieve the lower energy consumption of a processor. The offered frequencies and voltages can be insufficient to meet the energy consumption limit, therefore it might be necessary to throttle the current thread as well. The throttling mechanism to reduce the power consumption results in a performance degradation of a thread, even in the case of I/O-bound threads.

A thread's power consumption cannot only affect threads of its own priority-queue, but also threads of other priority-queues. Consider the following case: a low priority thread consumes so much power that it enforces its own throttling and also the throttling of the following higher priority threads. In this case, the higher priority thread suffers from the behavior of a lower priority thread, possibly resulting in priority inversion. Consequently, an energy-aware scheduling policy must prevent this behavior.

Fair Energy Partitioning To favor energy-efficient threads, energy-aware schedulers have to partition the system's energy but not the time among threads. For a fair energy partitioning, it is mandatory that each thread of a run-queue does not consume more energy than allowed during its quantum. We call the amount of energy a thread may consume during its quantum *thread energy budget*. It is defined as

$$E_{\mathrm{budget}_{\mathrm{thread}}} = P_{\mathrm{limit}} \cdot t_{\mathrm{timeslice}} \tag{3.1}$$

while P_{limit} is the pre-defined power consumption which a thread may not exceed and $t_{\mathrm{timeslice}}$ the timeslice length of the thread assigned by a scheduler. At the beginning of a thread's quantum, a scheduler set a thread's thread energy budget. With the thread's further execution, its energy budget decreases until it will be empty. If a thread's energy budget is empty, a scheduler will schedule the next thread and will reset the thread's energy budget.

To schedule the next thread directly after a thread has exhausted the energy of its energy budget, would required to account a thread's energy consumption continuously. Nevertheless, this is impossible because a thread's energy consumption can only be accounted if the kernel executes, but not while a thread executes. Therefore, during a thread's next quantum, the thread has to be charged for the energy it has consumed more than permitted in its last quantum.

A thread energy budget solely assures a fair energy partitioning by executing a thread until its energy budget is exhausted. It does not limit a thread's power consump-

tion. Consequently, the length of a thread's quantum depends on the power consumption caused during its execution. If a thread's average power consumption is below the power limit P_{limit}, its quantum will be longer than the thread's pre-defined timeslice. Otherwise, if a thread's average power consumption is beyond P_{limit}, its quantum will be shorter than its pre-defined timeslice. Only if its average power consumption is equal to the permitted one, its quantum is as long as its timeslice.

This is shown in Figure 3.1. In this example four threads are scheduled and executed until they have exhausted their energy budget. A thread's energy budget $L \cdot T$ is equal to the amount of energy required to execute a thread with the average power consumption L for its timeslice T.

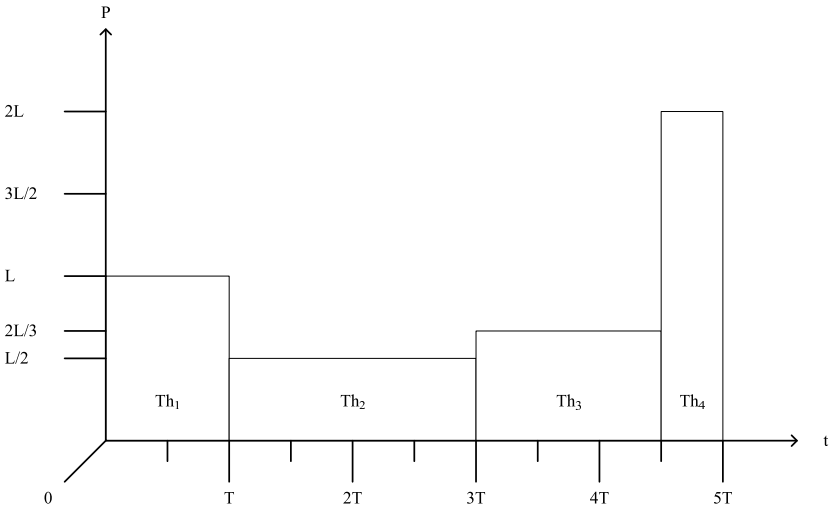

Figure 3.1: Thread Energy Budget

We want to fairly partition a processor's total energy among threads, thus we assign the processor's leakage energy [34] to a thread as well. Depending on the thread's power consumption, the thread spends a greater or lesser amount of its assigned energy for the processor's leakage energy. Threads with a low power consumption spend most of their energy for leakage, while threads with a high power consumption spend only a smaller fraction of their energy for leakage.

In order to prevent an exceeding of a processor's temperature above a threshold or to limit its energy consumption over a period of time, a scheduler has to limit the average power consumption of a thread during its execution. A thread energy budget is insufficient for this.

Structure In the next Section 3.2, we propose a design to limit a thread's power consumption without negatively affecting other threads and outline how to limit a thread's energy consumption. For a fair energy partitioning of a limited amount of energy, it is necessary to combine these two limits. We outline four different energy policies limiting a thread's power consumption and striving to achieve a fair energy partitioning by limiting the thread's power and energy consumption in Section 3.3. Thereby, two of the four policies permit threads having power consumptions above the permitted power limit to benefit from other threads having power consumptions beneath the limit. In Section 3.4, we propose a design of a per run-queue energy budget to permit

threads to benefit from other threads' power consumptions. We outline in the last Section 3.5 of this chapter a mechanism to preserve a thread's energy limit over several of its quanta, by charging the thread for its extra consumed energy.

3.2 Energy & Power Limit

Before we discuss how to assure a fair energy partitioning within an environment with limited energy resources or a critical processor temperature which a thread may not exceed, we introduce energy and power limits in the following two subsections. The energy limit's purpose is to assure a fair energy partitioning among threads. Therefore, a scheduler executes normally a thread until the thread has consumed the amount of energy permitted by the energy limit. We outline when a thread is preempted, although its energy budget is not empty. For restricting a processor's power dissipation an energy limit is insufficient, instead a power limit is required. It is responsible for limiting a thread's power consumption.

3.2.1 Energy Limit

The *energy limit* is defined analogously to a thread's energy budget (3.1). The only difference between the two is that it does not define how much energy a thread may still consume, instead it defines how much energy a thread may consume at most, before a scheduler schedules the next thread and refreshes the thread's energy budget.

When a scheduler assigns to a thread its energy budget, the energy budget is equal to the energy limit of this thread. The energy limit remains constant during the thread's execution, whereas the thread's energy budget decreases until it is exhausted. A scheduler will execute a thread no longer if the thread's energy budget is exhausted and the thread has consumed the amount of energy permitted by the energy limit, respectively.

As analyzed in the preceding section, a fair energy partitioning does no longer require a thread's timeslice to decide how long a thread is scheduled, instead the thread's energy limit and the thread's power consumption define how long its quantum lasts. Nevertheless, the timeslice assigned to a thread by its scheduling policy has a big impact, it is proportional to the thread energy budget (3.1) as well as to the energy limit.

Due to the fair energy partitioning, the length of a thread's quantum depends on the power consumption of a thread. A quantum has a lower as well as an upper bound. The maximum power consumption of a processor defines the lower bound, whereas a processor's idle power consumption defines the upper bound.

Threads having power consumptions below the power limit can decrease the system's reactivity. Their quanta will last longer than their timeslices if a scheduler does not split their quanta into smaller pieces. In order to avoid to split a quantum into smaller pieces and to assure a system's reactivity, it is possible to schedule the next thread. This has to happen at the latest after the current thread has executed for the duration of its timeslice, even if the thread's energy budget is not empty. A scheduler will discard the preempted thread's remaining energy of the thread's energy budget, but will refresh the thread's energy budget and quantum.

If a scheduler permits a thread to execute at most for the period of the thread's timeslice, although the thread's energy budget is not empty, we will call this a *non-strict energy limit*. Thus, a scheduler penalizes threads having power consumptions below the power limit in comparison to threads having at least power consumptions

equal to the permitted one, because threads with power consumptions below the power limit may not consume their assigned energy.

A fair energy partitioning demands each thread to execute until it has consumed the assigned energy of its energy budget, as long as it has not gotten preempted or blocked. As outlined before, this can decrease the system's reactivity. If – for a fair energy partitioning – we accept that a thread's quantum can be longer than a thread's timeslice, we will call this a *strict energy limit*.

We show the difference between the strict energy limit and the non-strict energy limit in Figure 3.1 and in Figure 3.2, respectively. Thread Th_1 executes in both cases for the time T and thread Th_4 for $\frac{T}{2}$. Applying the strict energy limit or the non-strict energy limit has only a consequence for the threads Th_2 and Th_3, because their power consumptions are below the power limit. If we apply the strict energy limit, thread Th_2's quantum will last for $2T$, thread Th_3's quantum for $\frac{3T}{2}$, otherwise each thread's quantum would only last for T.

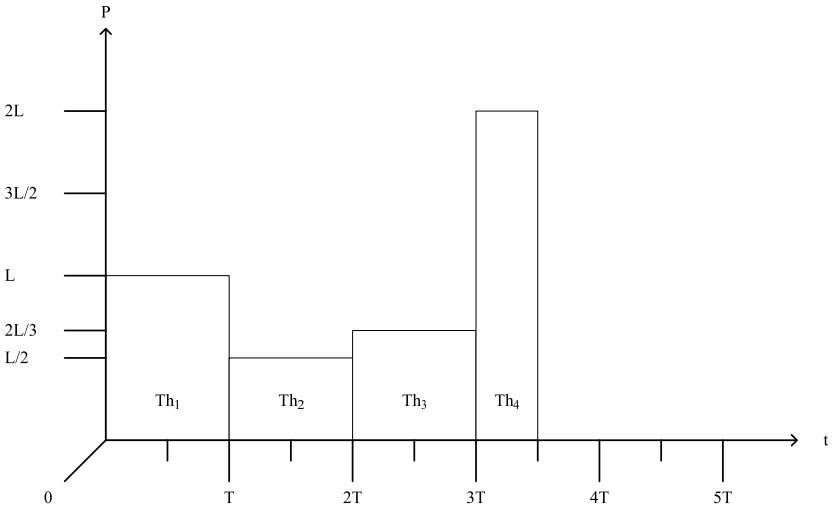

Figure 3.2: Non-Strict Energy Limit

Strict vs. Non-Strict Energy Limit Next, we compare the strict energy limit and the non-strict energy limit and outline their implications on a fair energy partitioning.

The advantage of a strict energy limit is a fair energy partitioning of the processor's total consumed energy among the threads, whereas the disadvantage may be a reduced system reactivity due to threads having longer lasting quanta than timeslices. In contrast to the strict energy limit, the non-strict energy limit bounds a thread's quantum by a thread's timeslice. Therefore, it does not suffer from the problem of a reduced system reactivity like the strict energy limit. Its disadvantage is an unfair energy partitioning among all threads, due to the limited execution time of a thread's timeslice.

3.2.2 Power Limit

As mentioned before, a thread energy budget and its related energy limit are inadequate for limiting a processor's power consumption. Therefore, we require a power limit.

In contrast to an energy limit which aims at partitioning a processor's energy fairly among the threads, a power limit's aim is to restrict a processor's power consumption. Mechanisms like dynamic frequency and voltage scaling can achieve this. They decrease the maximum power consumption of a processor, but the frequencies and voltages cannot be scaled continuously – instead the hardware offers only a few. This prohibits to meet an arbitrary power limit. We require the more fine-grained instruction throttling mechanism to limit a processor's as well as a thread's average power consumption in order to meet an arbitrarily pre-defined power limit.

A kernel cannot use throttling for limiting the maximum power consumption of a processor. It only reduces the processor's number of duty cycles. During a duty cycle [12] a thread may consume the maximum power. Nonetheless, throttling permits to limit the average power consumption of a thread during its quantum, to meet the pre-defined power limit. For this thesis we consider instruction throttling as the mechanism to limit a thread's average power consumption in order to meet a pre-defined power limit.

If we loose the requirement of meeting a pre-defined power limit over the period of a thread's quantum, and simply enforce to meet it over a hyper-period of several threads' quanta, a scheduler can *transfer* energy between threads. The transferred energy results from threads having power consumptions below the power limit.

Without permitting an energy transfer, a thread's average power consumption caused during its quantum must not exceed the power limit. Therefore, we call it a *strict power limit*. If a thread may exceed the power limit due to performed energy transfers, we call it a *non-strict power limit*. A strict power limit enforces to meet the pre-defined power limit over the period of a thread's quantum, and the non-strict power limit over the hyper-period of several threads' quanta.

Strict vs. Non-Strict Power Limit After having introduced the strict power limit and the non-strict power limit, we discuss their implications on a thread's performance.

The advantage of the non-strict power limit is the permitted energy transfer between threads offering energy and threads having power consumptions beyond the power limit. Energy transfers can reduce a thread's throttling, in the best case they can even avoid a thread's throttling. The offered energy results from threads having power consumptions beneath the power limit. Consequently, a non-strict power limit can increase the system's performance significantly in comparison to a strict power limit. For exhausting a processor's power limit over a hyper-period, a scheduler needs to transfer energy. Without performing energy transfers, threads having power consumptions below the power limit cause a processor's average power consumption beneath the power limit.

To outline the difference between these two limits, we consider their impacts on our first example (cf. Figure 3.1). Although thread Th_4 has a power consumption of about $2L$, it consumes solely L due to the throttling mechanism, which we apply because of the strict power limit, as shown in Figure 3.3. By contrast, the non-consumed energy of threads Th_2 and Th_3 will nearly avoid the throttling of thread Th_4, if we apply the non-strict power limit. We show this in Figure 3.4.

We discuss the two proposed power limits in detail in Appendix A. There we present how one can determine the amount of energy E_{offered} a thread can offer, and how a scheduler can assure that a thread receives at most its assigned fraction of offered energy E_{frac}.

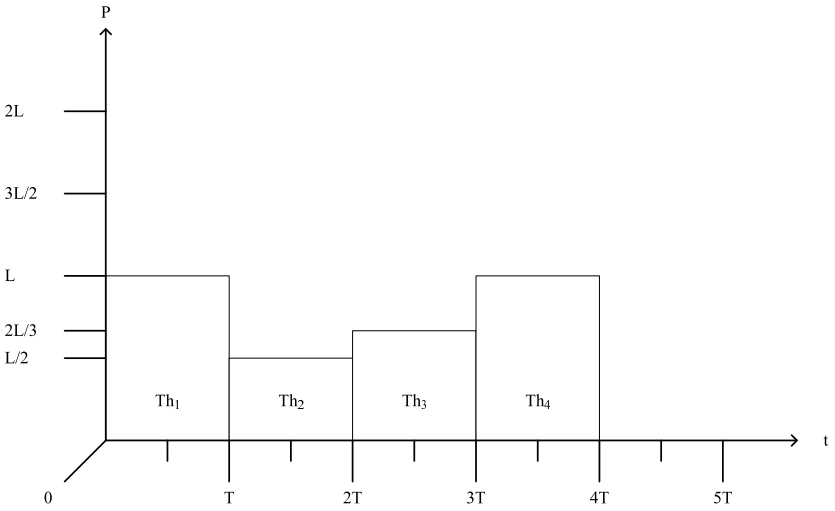

Figure 3.3: Strict Power Limit

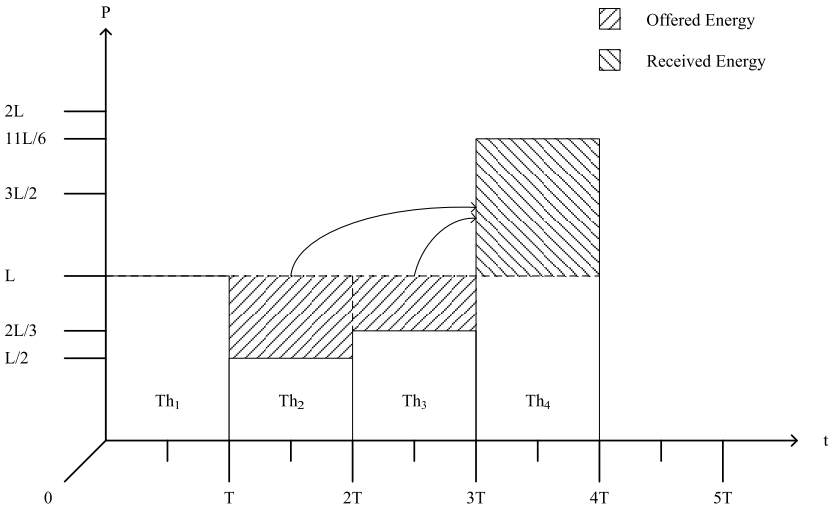

Figure 3.4: Non-Strict Power Limit

3.3 Energy Policies

In the next section, we explain how both the strict and non-strict energy and power limits can be combined to apply energy-aware scheduling policies, in order to realize a fair energy partitioning while limiting the processor's power consumption. Besides, we discuss what their advantages and disadvantages are in comparison to one other. The ensuing section deals with the realization of a fair partitioning of the offered energy in the case of a non-strict power limit.

We have introduced strict as well as non-strict energy and power limits in the antecedent section. Both limits can be combined with each other, because

- the power limit merely assures that a thread does not exceed a pre-defined power limit in order to restrict a processor's power dissipation, and

- the energy limit only assures a fair energy partitioning among threads, but does not have an influence on the course of a thread's power consumption.

Thereby, it is unimportant whether an energy policy meets the limits strictly or not. In the next four subsections, we discuss the four different possible combinations, and how fair they can partition the system's energy. Additionally, we consider the energy policies' implications on a thread's runtime.

3.3.1 Strict Power & Non-Strict Energy Limit

We have outlined in Subsection 3.2.2 that a thread's average power consumption may not exceed the power limit in the case of a strict power limit. Therefore, an energy transfer among threads is not possible. The non-strict energy limit enforces that a non-blocking thread's quantum may last at most as long as its timeslice lasts. Without considering the power limit, the maximum power dissipation of a processor defines its lower bound (cf. Subsection 3.2.1). Due to the limited power consumption caused by the strict power limit, its lower bound is equal to its upper bound. Both bounds are equal, because a thread can have consumed its assigned energy at the earliest after a scheduler has executed it for the period of its timeslice due to the strictly preserved power limit. This energy policy is equivalent to a scheduling policy throttling each thread individually and executing a thread at most for the period of its timeslice.

A scheduler can avoid to account the energy consumption of a thread on each timer tick; it is sufficient to execute the thread until its timeslice expires. In the case of threads having average power consumptions below the allowed power limit, their energy budget is never exhausted by themselves. Due to these threads and the strict power limit forbidding energy transfers, threads cannot exceed a processor's power limit.

This policy enforces that a thread can at the earliest have consumed its assigned energy after the period of its timeslice and may execute at most for the period of its timeslice. Thus, a scheduler can avoid to account a thread's consumed energy. Nonetheless, to ascertain whether a scheduler must throttle the thread at the beginning of the thread's next quantum to charge it for exceeding the power limit in its previous quantum, it is necessary to account a thread's power consumption after its quantum.

Referring to our example, thread Th_4 must be throttled assuring an average power consumption equal to the permitted one of L. It cannot benefit from the lower power consumptions of the threads Th_2 and Th_3, because we do not allow energy transfers. Due to the non-strict energy limit, the quanta of the two latter threads are equal to their timeslices. Which we have outlined in Figure 3.5.

The strict power limit and the non-strict energy limit only assure that a thread does not suffer from the energy consumption caused by the execution of other threads. It is insufficient for a fair energy partitioning among threads, this can only be achieved by applying the strict energy limit discussed in the next subsection.

3.3.2 Strict Power & Strict Energy Limit

Analogously to the previous energy policy, this energy policy preserves the pre-defined power limit strictly. Consequently, the lower bound of a thread's quantum is its timeslice length. Before, a thread strictly preserving the power limit cannot have consumed its assigned energy. In contrast to the previous energy policy, the idle power consumption defines the upper bound. The upper bound depends on a thread's energy budget, as outlined in Subsection 3.2.1.

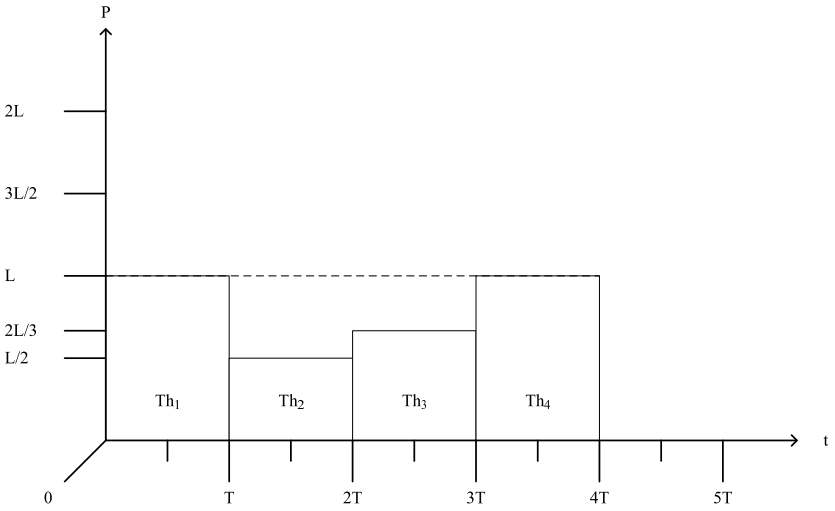

Figure 3.5: Strict Power & Non-Strict Energy Limit

In this way, a fair energy partitioning among threads is possible, whereas a scheduler prohibits energy transfers. The drawback of the strict energy limit is the extended quantum in comparison to the timeslice length of threads having average power consumptions below the power limit. Thus, the system's reactivity decreases and the throughput of threads having quanta lasting as long as their assigned timeslices drops as well. By contrast, it is possible that threads can increase their throughput which have average power consumptions below the power limit.

In comparison to the remaining policies, threads which have average power consumptions beyond the power limit, suffer at most from this policy. They cannot benefit from power consumptions below the power limit of other threads by an energy transfer. A scheduler executes them less frequently within a period of time, because of quanta lasting longer than timeslices. Therefore, this energy policy causes the worst performance for these threads, which we show in Figure 3.6. We discuss the influence on threads having extended quanta later on.

3.3.3 Non-Strict Power & Non-Strict Energy Limit

This energy policy and the next energy policy consider the case of a non-strict power limit and its impacts on strict and non-strict energy limits. Contrary to the energy policy outlined in Subsection 3.3.1, where a scheduler cannot transfer non-consumed energy to other threads, this policy allows energy transfers. We have pointed out in Subsection 3.2.2 that it is mandatory for exhausting a processor's power limit to transfer energy between threads. This energy transfer can have two different semantics:

1. Non-consumed energy is given away to other threads.
2. Non-consumed energy can be used to give other threads the permission to exceed the power limit, but not their thread energy budgets.

We call the first semantics *extended energy budget*, because the received energy is given away to a thread and extends the thread's energy budget. In contrast to that, we call the

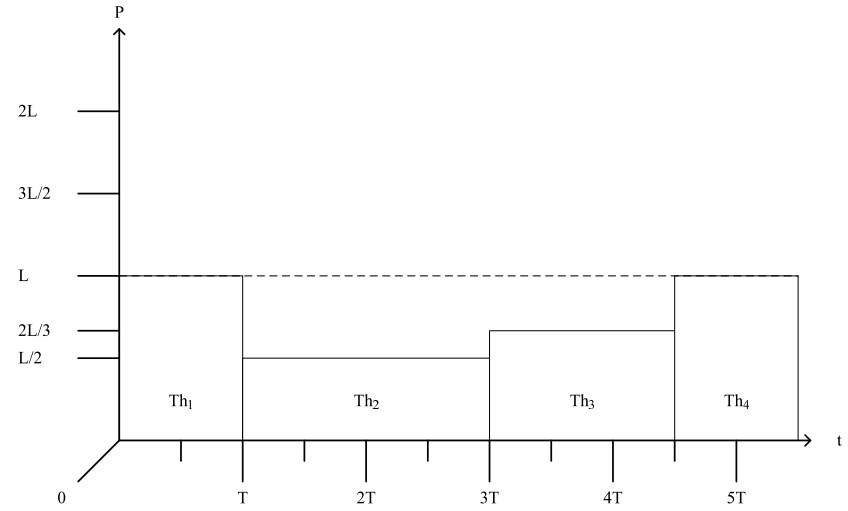

Figure 3.6: Strict Power & Strict Energy Limit

second semantics *exceeded power limit*, because it merely permits a thread to exceed the power limit, but not its energy budget.

Extended Energy Budget In the case of giving the non-consumed energy away, the energy neither extends nor reduces an energy receiving thread's quantum, because the energy permits a thread to exceed the power limit during its quantum, but not to extend its quantum. Therefore, a thread's quantum lasts as long as in the case of non-performed energy transfers.

A thread receiving energy consumes more energy during a period of time than a thread offering energy, because the latter thread gives its offered energy away. This applies also for the case of the strict power limit and the non-strict energy limit. The extended energy budget energy transfer, however, even increases the energy gap between threads with power consumptions below the limit and others above even more, because only threads receiving energy can benefit from it. A thread receiving offered energy can consume the energy of its thread energy budget, and additionally it can consume its fraction of the offered energy E_{frac} for exceeding the power limit. Hence, this policy in combination with the extended energy budget energy transfer results in the best performance for the latter threads with respect to the remaining proposed policies.

Exceeded Power Limit Contrary to giving away the non-consumed energy, the second semantics only gives other threads the permission to exceed the power limit. A thread can exceed the power limit, until it has spent its complete fraction of offered energy E_{frac} for the energy caused by the power consumption above the power limit. The fraction of offered energy E_{frac} a thread receives does not extend the thread's energy budget. Consequently, a thread's quantum will not last as long as its timeslice if its average power consumption is beyond the limit due to energy transfers. The advantage of the exceeded power limit energy transfer is the diminished energy gap between threads offering and threads receiving energy.

Independently of an energy transfer's semantics, it is necessary to account a thread's energy consumption in order to determine when a thread has consumed its assigned fraction of offered energy E_{frac}. If we apply the exceeded power limit energy transfer, a thread's lower bound of its quantum will no longer depend only on its timeslice, but also on the processor's maximum power consumption. A thread's quantum length still depends on its timeslice, because a thread's energy budget is proportional to its timeslice (3.1).

We outline the difference between these two semantics in the following two figures. Therefore, we consider once again our initial example outlined in Section 3.1. If we apply the extended energy budget energy transfer, the offering threads cannot benefit from their offered energy (cf. Figure 3.7). The offering threads can benefit from their offered energy if we apply the exceeded power limit energy transfer, because the turn around time will be decreased (cf. Figure 3.8).

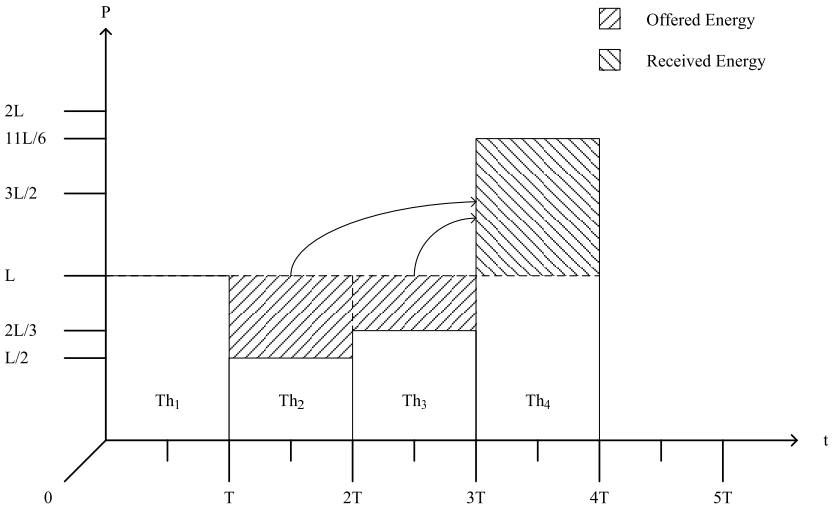

Figure 3.7: Non-Strict Power & Non-Strict Energy Limit - Extended Energy Budget

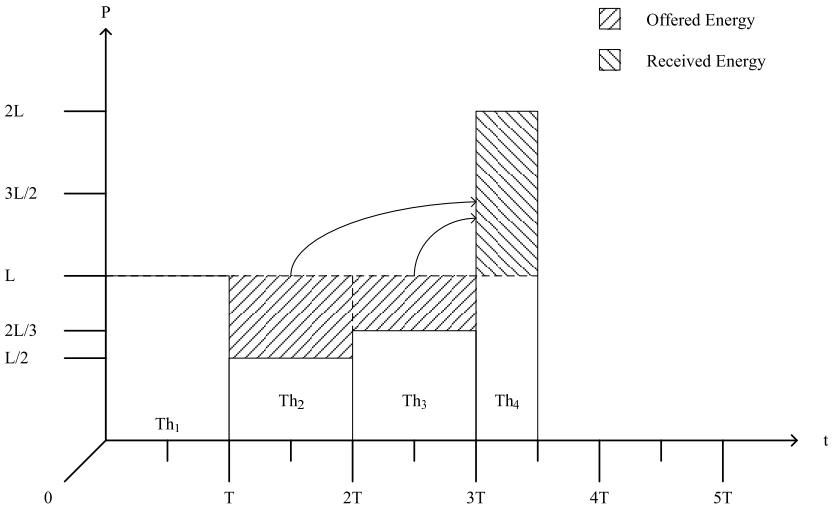

Figure 3.8: Non-Strict Power & Non-Strict Energy Limit - Exceeded Power Limit

3.3.4 Non-Strict Power & Strict Energy Limit

In contrast to the energy policy realizing the strict power limit and the strict energy limit outlined in Subsection 3.3.2, this energy policy permits energy transfers. In this subsection we outline which thread may receive the energy, and discuss the implications of our four proposed energy policies on the performance of threads receiving and threads offering energy.

The non-strict energy limit of the preceding policy forbids to transfer energy to threads not exceeding the power limit due to the restricted quantum length of a thread's timeslice. In contrast to that, the strict energy limit of this policy allows to transfer energy to threads not exceeding the power limit by extending their quanta. Nonetheless, such an energy transfer has the following drawbacks:

- A thread's quantum will no longer have an upper bound or the upper bound will solely be defined if the amount of received energy is limited. Consequently, the turn around time of a thread within a run-queue may increase steadily, if a thread's quantum has no upper bound as presented in Figure 3.9.
- The energy transfer is no longer only used for exhausting a processor's power limit, instead it is additionally used for extending a thread's quantum, but this is not the intention of the energy transfer.

Next, we show how an energy-aware scheduler can realize a fair energy transfer without these drawbacks.

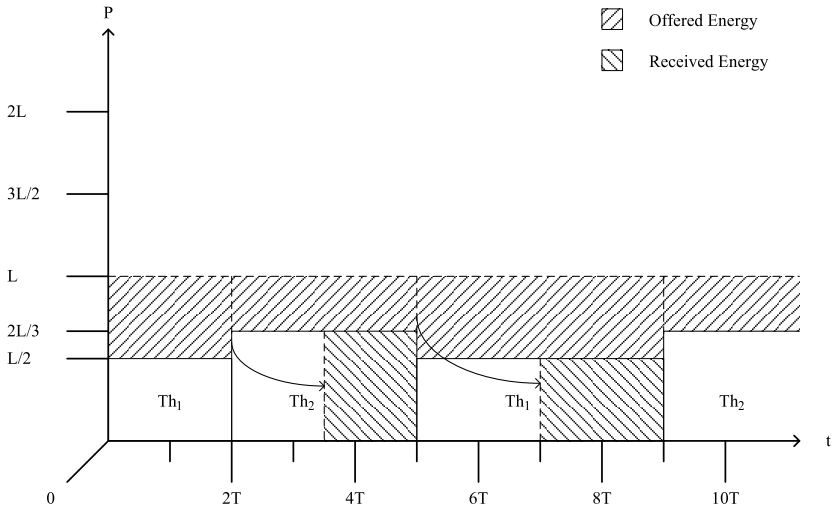

Figure 3.9: Energy Transfer Permitting Unbound Turn Around Time

We have outlined in the last subsection two different ways of performing an energy transfer. The energy can be given away to a thread or merely be used for permitting a thread to exceed the power limit. This applies also for this policy. The non-strict energy limit of the previous policy inhibits a fair energy partitioning, even in the case of applying the exceeded power limit energy transfer. This policy, however, can achieve this by applying the exceeded power limit energy transfers.

Extended Energy Budget The extended energy budget energy transfer prevents threads offering energy to other threads to benefit from their offered energy. This is the case because a scheduler does not decrease the quanta of threads receiving energy. These non-decreased quanta are the reason why the performance of threads only offering and not receiving energy remains unchanged in comparison to their performance while prohibiting an energy transfer. Nevertheless, energy transfers result in an increase of the system's throughput caused by the threads receiving energy in comparison to the energy policy realizing the strict power limit as seen in Figure 3.10.

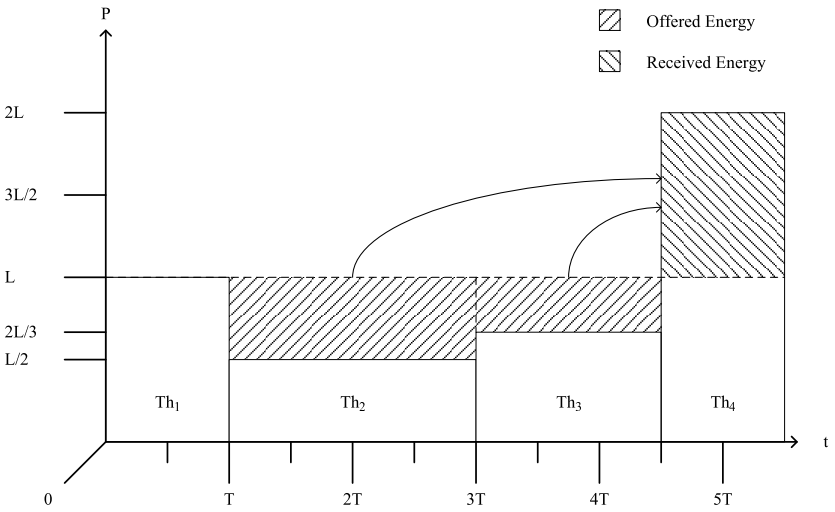

Figure 3.10: Non-Strict Power & Strict Energy Limit - Extended Energy Budget

Exceeded Power Limit This policy in combination with the exceeded power limit energy transfer assures a fair energy partitioning like the policy introduced in Subsection 3.3.2, which realizes the strict power limit and the strict energy limit. The advantages of this policy are its permitted energy transfers to increase the system's throughput. The increase is achieved on no thread's account. Threads having power consumptions below the power limit and other having power consumptions beyond the power limit can both benefit from this policy as long as they apply the exceeded power limit energy transfer.

Threads offering energy to other threads benefit from this policy, because a scheduler diminishes the quanta of threads receiving the offered energy. Thus, a scheduler executes them more frequently within a period of time and their throughput increases. The same applies to threads receiving energy. Although their quanta are shorter, they can make the same progress as during their non-reduced quanta. Moreover, they can normally achieve more progress during their quanta. They exhaust a smaller fraction of their thread energy budgets for a processor's leakage power. We have illustrated this scenario in Figure 3.11.

Energy Policy Comparison After we have outlined the design of the four policies and have already discussed the impacts of this policy on the performance of threads which may – if allowed – offer or receive energy, we proceed with the discussion of

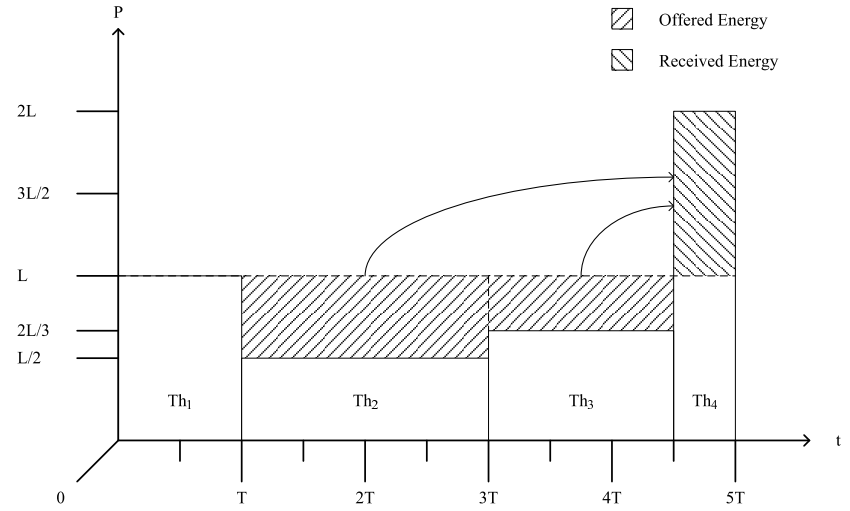

Figure 3.11: Non-Strict Power & Strict Energy Limit - Exceeded Power Limit

the three previously introduced policies. For our considerations, we examine a set of threads consisting of both types.

At first, we have proposed the strict power limit and the non-strict energy limit energy policy. This policy forbids energy transfers and restricts a quantum length to a timeslice, hence threads of both types suffer from this policy. This policy results in the worst performance for threads offering energy, because it limits a thread's quantum length and prohibits energy transfers. Due to the prohibited energy transfer, the performance of threads receiving energy is not as good as in the case of permitted energy transfers, but not as bad as in the case of the second energy policy, which realizes the strict power limit as well as the strict energy limit. Threads which receive energy suffer from this policy because of the extended quanta of threads which offer energy. Therefore, they will have the worst performance if we apply this policy. Contrary, the longer quanta of threads offering energy can improve their performance in comparison to the first policy.

The last two proposed policies allow energy transfers, therefore the performance for both types of threads is at least as good as without energy transfers but often even better. In the case of the non-strict power limit and the non-strict energy limit, threads receiving energy can achieve their best performance if we apply the extended energy budget energy transfer. This policy does not consider their received energy. Their performance is not as good as if we apply the exceeded power limit energy transfer, but even better than with prohibited energy transfers. Threads offering energy can achieve the same performance as in the case of the strict power limit if we apply the extended energy budget energy transfers, because they cannot benefit from their transfers. If we apply the exceeded power limit energy transfer, their performance will improve because a scheduler will execute them more frequently due to the shorter quanta of threads receiving energy.

3.4 Run-Queue Energy Budget

In the antecedent section, we have outlined how a fair energy partitioning can be realized. Thereby, we have only considered in which way each thread receives its assigned amount of energy, but we have not discussed what is required to fairly partition offered energy. Within this section we propose a design for a per run-queue energy budget permitting to fairly partition offered energy among threads. Before, we outline why a first come first serve scheme is insufficient.

3.4.1 Naïve Solution

We have pointed out in the previous section that the semantics of an energy transfer determines whether we can fairly partition energy among all threads or whether it leads to an increased unfairness between threads offering energy and threads receiving energy. If we consider that a scheduler can fairly partition the system's energy among threads, a first come first serve scheme might result in an unequal performance of threads with the same priority and weight, respectively, as well as an equal power consumption, even in the case of permitted energy transfers. An example for these threads are several instances of the same task. The unequal performance results from throttling one instance of the task more frequently than another instance. The more frequently throttled task spends more of its energy for the idle energy consumed during its throttling than the other one.

In the following, we consider the case of a set of threads consisting of both threads offering and receiving energy. A scheduler transfers the offered energy in a first come first serve scheme. Thus, each thread willing to receive energy may receive its requested amount of energy until the offered energy is exhausted. Besides, we consider that the amount of offered energy is insufficient to avoid a throttling of each thread, a scheduler must at least throttle one thread.

Extended Energy Budget At first, we consider that the offered energy is given away in a first come first serve scheme. A scheduler does not have to throttle threads receiving their requested energy. These threads' quanta last as long as their timeslices. They can achieve their best performance during their quanta. Threads requesting offered energy which a scheduler schedules after the offered energy has been exhausted cannot consume as much energy as the threads before. In addition, a scheduler must throttle them due to the non-received energy. Hence, they cannot achieve as much as the previously scheduled threads during their quanta. Consequently, a first come first serve scheme is insufficient for a fair energy partitioning, which strives to ensure the same performance for each instance of a task.

Exceeded Power Limit This will be also the case if we apply the exceeded power limit energy transfer and permit threads to receive the energy in a first come first serve scheme. The threads receiving energy and therefore avoiding their throttling can realize their best performance during their quanta, although their quanta are shorter than their timeslices. After the first threads have exhausted the offered energy, a scheduler must throttle the remaining threads requesting offered energy. The quanta of the latter threads last as long as their timeslices, but they have to spend a fraction of their assigned energy for their throttling. This prevents that they can make the same progress as threads which have received the offered energy resulting in an unfair partitioning

of offered energy. The performance gap between threads receiving energy and threads being throttled results from throttling the latter threads. In the case of the extended energy budget energy transfer the gap is even larger, because the throttled threads receive less energy than the energy receiving threads.

To confirm this, we consider a setting of three threads. The first thread Th_1 offers the two remaining threads $\mathrm{Th}_{2'}$ and $\mathrm{Th}_{2''}$ – instances of thread Th_2 – its non-consumed energy. We show this in Figure 3.12. These two latter threads have a power consumption of $2L$. The offered energy is insufficient to fulfill $\mathrm{Th}_{2'}$'s request, therefore its average power consumption is only $\frac{3L}{2}$ and it must be throttled. Thread $\mathrm{Th}_{2''}$ has only an average power consumption of L, because after $\mathrm{Th}_{2'}$'s execution no more offered energy was left.

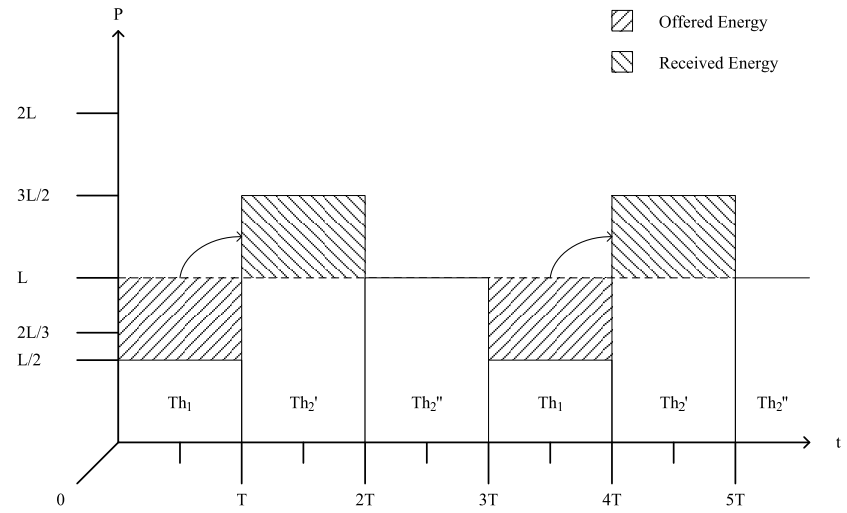

Figure 3.12: Naïve Run-Queue Energy Budget - Extended Energy Budget

As mentioned before, if we apply the exceeded power limit energy transfer, the unfairness will result from throttling one thread and not the other one. We present this in Figure 3.13.

3.4.2 Proposed Solution

The run-queue energy budget proposed in this subsection solves the problems outlined in the previous subsection. Our proposed run-queue energy budget assures a fair energy partitioning of offered energy.

As discussed before, a simple first come first serve scheme is insufficient for a fair partitioning of offered energy. An obvious solution for this problem is to partition the offered energy among all threads of a run-queue or priority queue. Thereby, for proportional share scheduling policies one has to consider the amount of tickets or the weights of threads to partition the energy fairly. The drawback of this policy is the coarse-grained partitioning of offered energy. Therefore, some threads can get a fraction of the offered energy, although they do not require it. Consequently, the system's performance cannot be as good as if a scheduler fairly partitions the energy

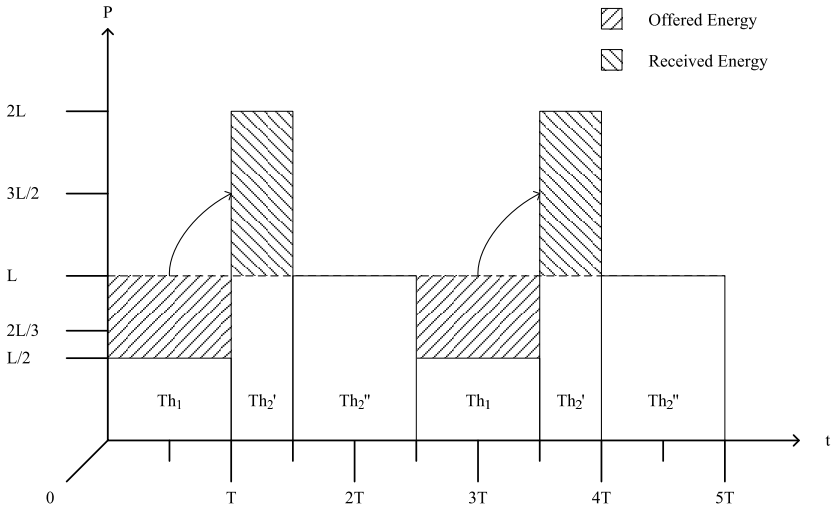

Figure 3.13: Naïve Run-Queue Energy Budget - Exceeded Power Limit

only among threads requiring it. The same will happen if a thread receiving energy cannot exhaust its fraction of offered energy.

The idea of the proposed per run-queue energy budget is that after a thread has executed for its quantum, it defines for how many threads the offered energy lasts and how much of the offered energy each thread may receive at most. Our proposed energy budget has the following structure:

- E_{frac}
- threads
- $E_{\text{non-consumed}}$

The energy E_{frac} defines the amount of energy each thread may receive at most and threads defines the remaining number of threads permitted to receive energy. In the case of proportional share scheduling policies, threads defines the sum of tickets or the sum of weights of threads allowed to receive energy. If a thread has not consumed its complete energy E_{frac}, a scheduler adds the remaining energy to the energy $E_{\text{non-consumed}}$. A scheduler can partition this energy among threads later on.

Independently of any scheduling policy, the total amount of offered energy which a scheduler can still transfer is defined as:

$$E_{\text{total offered}} = E_{\text{frac}} \cdot \text{threads} + E_{\text{non-consumed}} \tag{3.2}$$

After we have presented the structure of a per run-queue energy budget, we outline how it assures a fair energy transfer. We have to distinguish the following three cases:

1. A thread exhausts its complete fraction of offered energy.
2. A thread receives a share of its fraction of offered energy.
3. A thread offers its non-consumed energy.

Case 1 If a thread has exhausted its complete fraction E_{frac} of the offered energy, a scheduler will decrement threads to reflect the thread's energy consumption. In the case of a proportional share scheduling policy, a thread will have consumed the energy $E_{\text{frac}} \cdot \text{tickets}_{\text{thread}}$, while $\text{tickets}_{\text{thread}}$ is the number of tickets held by a thread. Therefore, a scheduler will decrement threads about $\text{tickets}_{\text{thread}}$.

Case 2 If a thread has not consumed the energy of its complete fraction E_{frac} and $E_{\text{frac}} \cdot \text{tickets}_{\text{thread}}$, respectively, a scheduler must once again partition the remaining energy among the threads. One possibility is to partition this energy among the remaining threads having not already received their fractions of the offered energy. The drawback of this approach is the dependency of a thread's fraction E_{frac} on the scheduling sequence. This makes it impossible to fairly partition the offered energy. To prevent this, the update of E_{frac} must be delayed until threads is zero. This permits to accumulate the non-consumed energy of several threads' fractions. The value threads will be zero if the number of predicted threads willing to receive the offered energy $\text{threads}_{\text{receive}}$ have received their energy. If a thread has consumed less of the offered energy than permitted $E_{\text{received}} < E_{\text{frac}}$, the non-consumed energy $E_{\text{non-consumed}}$ will be updated:

$$E_{\text{non-consumed}} = E_{\text{frac}} - E_{\text{received}} + E_{\text{non-consumed}} \tag{3.3}$$

Afterwards, if the threads which have received the offered energy have caused a scheduler to decrement threads to zero, a scheduler can partition the non-consumed energy $E_{\text{non-consumed}}$ of threads' fractions among $\text{threads}_{\text{receive}}$ threads.

Case 3 If a thread has an average power consumption below the power limit, it can offer its non-consumed energy E_{offered} to other threads. A scheduler can fairly partition the energy and, in addition to that can fairly partition the non-consumed energy $E_{\text{non-consumed}}$ if threads is zero. The fraction of the offered energy a thread may receive is:

$$E_{\text{frac}} = \frac{E_{\text{offered}} + E_{\text{non-consumed}}}{\text{threads}_{\text{receive}}} \tag{3.4}$$

while a scheduler has reset threads to the number of threads $\text{threads}_{\text{receive}}$ which may receive the energy.

Due to the scheduling sequence or several threads offering energy, threads can be greater than zero. In this case, a scheduler has to increase the non-consumed energy $E_{\text{non-consumed}}$ by E_{offered}, so that it can fairly partition the energy later on.

After we have outlined how our run-queue energy budget assures a fair energy transfer, we discuss under which conditions we can update or avoid the update of the energy E_{frac}. Afterwards, we present two figures outlining that our run-queue energy budget applies to both semantics of an energy transfer.

A thread receiving only a share of its assigned fraction of the offered energy can update the energy E_{frac}. Additionally, it can update this energy if its average power consumption is below the power limit. To avoid updating E_{frac} twice, a scheduler will solely update the non-consumed energy $E_{\text{non-consumed}}$ if a thread receives energy but does not consume its complete fraction.

Furthermore, a scheduler can update E_{frac} not only if threads is zero, but also if $\text{threads}_{\text{receive}} < \text{threads}$ or both are equal and $E_{\text{non-consumed}}$ is not zero. In the two latter cases, a scheduler can offer the non-consumed energy to the receiving threads, because it does not induce an unfair energy transfer.

Our run-queue energy budget is independent of the semantics of the energy transfer itself. It permits a fair partitioning of the offered energy for the extended energy budget energy transfer shown in Figure 3.14 as well as for the exceeded power limit energy transfer presented in Figure 3.15.

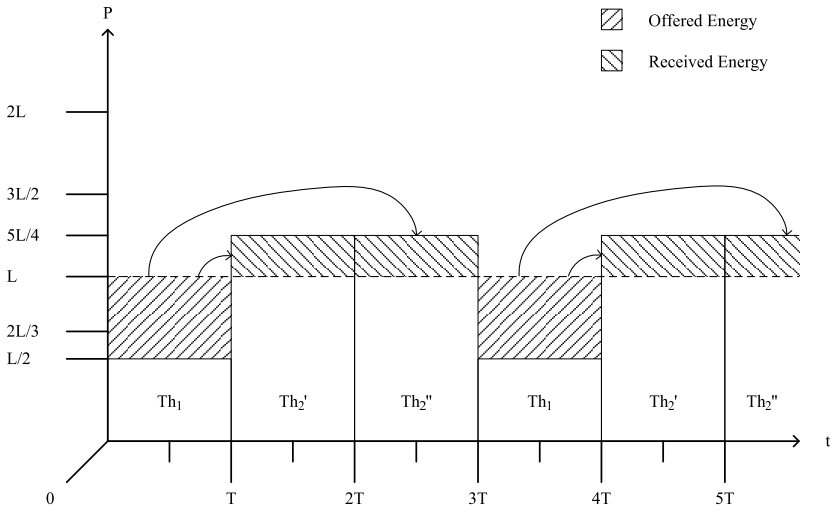

Figure 3.14: Proposed Run-Queue Energy Budget - Extended Energy Budget

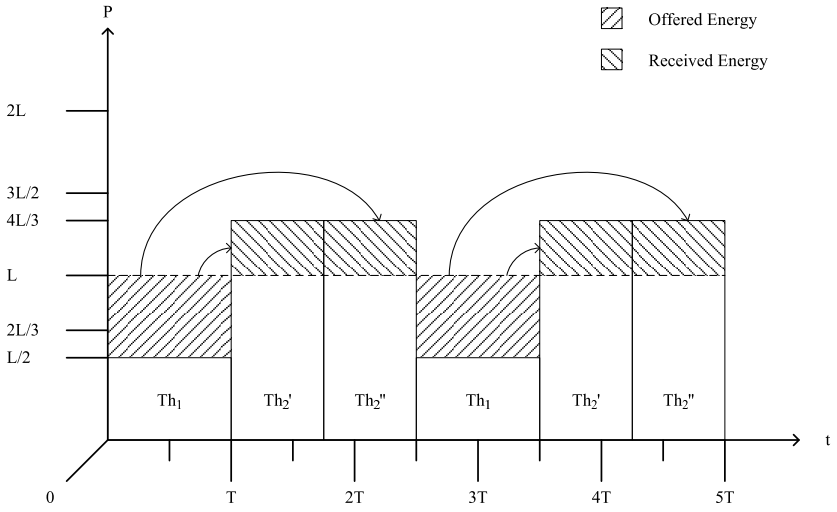

Figure 3.15: Proposed Run-Queue Energy Budget - Exceeded Power Limit

For the design of our run-queue energy budget, it is unimportant how many threads may receive the energy. The amount of offered energy, however, a thread may receive depends on the number of threads among a scheduler partitions the offered energy. Consequently, restricting the number of threads which may receive the energy can avoid throttling of threads requiring the non-consumed offered energy. Within the following part of this subsection, we discuss how a scheduler can estimate the number of receiving threads.

3.4.2.1 Receiving Threads

We have mentioned before that a scheduler can partition the offered energy among all threads of a run-queue. Next, we discuss what problem arises from a too coarse partitioning of offered energy. Afterwards, we outline how a scheduler can estimate the number of threads willing to receive the offered energy to partition the offered energy only among this number of threads.

Coarse Partitioning A scheduler can partition the offered energy among all threads of a run-queue. This is a drawback, in particular, for priority based schedulers. Often only a subset of threads is eligible to be scheduled due to the threads' priorities. Nonetheless, even partitioning the energy among all threads of a priority-queue belonging to a run-queue may be a too coarse partitioning, because not all of these threads require to receive energy.

The consequence of a too coarse partitioning is a smaller size of E_{frac} than necessary. Thus, a scheduler may throttle threads more often than necessary if the threads' required power consumptions to achieve their best performances cannot be satisfied. Later on we will prove in Appendix B, that a scheduler will fulfill each thread's request at the latest after n^2 runs of the n threads of a run-queue if the offered energy is sufficient to fulfill the requests. Even if each thread's request can be fulfilled after n^2 runs, it will be an advantage if a scheduler only partitions the offered energy among threads which are eligible to be scheduled and require the offered energy.

Fine Partitioning To estimate the number of threads $\text{threads}_{\text{receive}}$ requiring the energy, it is sufficient that a scheduler accounts how many threads it has or would have throttled during the last execution of the scheduled threads. A scheduler would have throttled a thread, if the thread's estimated power consumption had been beyond the power limit, but it had not throttled the thread due to energy transfers.

For proportional share scheduling policies, it is insufficient to account only the number of threads requiring the energy, instead a scheduler must account the threads' tickets or weights. Consequently, a thread has to decrement $\text{threads}_{\text{receive}}$ about its number of tickets and its weight, respectively. The energy $E_{\text{non-consumed}}$ will only be made available if threads is zero or equal to $\text{threads}_{\text{receive}}$. Hence, an energy transfer will be impossible if threads is smaller than any receiving thread's weight and its number of tickets, respectively. Therefore, a receiving thread may consume the offered energy, even if threads is smaller than its assigned weight or number of tickets. Consequently, the fraction of offered energy a thread holding $\text{tickets}_{\text{thread}}$ tickets may receive is:

$$E_{\text{frac}} = min\{E_{\text{budget}_{\text{thread}}} \cdot \text{threads}, E_{\text{budget}_{\text{thread}}} \cdot \text{tickets}_{\text{thread}}\} \tag{3.5}$$

After the thread has executed for its quantum and if threads is smaller than its number of assigned tickets, it must not decrement tickets about $\text{tickets}_{\text{thread}}$, instead it must reset it to zero.

We have explained that it is unnecessary to partition the offered energy among all threads of a run-queue, because a scheduler can efficiently estimate how many threads will probably require the offered energy.

3.4.2.2 Handling Preemptions

Next, we discuss when we have to update E_{frac} and $\text{threads}_{\text{receive}}$ to realize a prioritized energy transfer to fulfill the demands of a priority based scheduling policy. Afterwards, we outline how to assure that the run-queue energy budget reflects the offered energy after a thread's preemption and before the preempted thread continues with its execution.

Priority based scheduling policies require a prioritized energy transfer. A prioritized energy transfer partitions the offered energy only among the threads eligible to run belonging to the highest priority and probably requesting the offered energy. Runnable threads assigned to lower priorities and requesting the offered energy are not considered by a scheduler. Only if the lower priority threads are eligible to run, a scheduler will consider them.

A priority based scheduling policy requires to update $\text{threads}_{\text{receive}}$ as well as E_{frac}, whenever the priority of the previously executed thread is unequal to the priority of the next scheduled thread. The offered energy is solely partitioned among threads of a priority-queue willing to receive the energy. If no thread is willing to receive the energy, we will set $\text{threads}_{\text{receive}}$ to one, in order to give a thread the opportunity to receive the offered energy. This permits to realize a prioritized energy transfer. Without a prioritized energy transfer, the number of lower priority threads can affect the fraction a high priority thread can receive of the offered energy.

The main disadvantage of a priority based scheduler is the unfair partitioning of offered energy among eligible threads of a priority-queue. This will be the case if a scheduler has scheduled a few lower priority threads having already received their complete fraction and a scheduler has not already scheduled the remaining threads of a priority-queue. Moreover, if these latter threads are scheduled after a higher priority thread having received an amount of the offered energy, they cannot receive the fraction of the offered energy previously designated to them. In the worst case, the higher priority thread has received the complete offered energy and the lower priority threads which have not yet received a share of this energy cannot receive any offered energy anymore, because it is exhausted.

As preliminarily pointed out in Subsection 3.4.2, the invariant of our proposed run-queue energy budget structure is the definition of the energy which is left for subsequent energy transfers after a thread's execution. Next, we outline how we assure that the run-queue energy budget reflects the offered energy, which a scheduler can transfer after a thread's execution. We have to consider the following two cases:

1. A thread gets preempted during its quantum.
2. A thread continues with its execution after another thread has preempted this thread.

Case 1 To meet the requirement that the run-queue energy budget defines the amount of energy which can be transferred, it is not sufficient to define $E_{\text{frac}_{i+1}}$ as follows: [1]

$$E_{\text{frac}_{i+1}} = \frac{E_{\text{frac}_i} \cdot \text{threads}}{\text{threads}_{\text{receive}}} \tag{3.6}$$

E_{frac_i} is the offered energy before the update, and $E_{\text{frac}_{i+1}}$ is the offered energy after the update. The reason for this is that a preempted thread may have received energy of

[1] for clarity, we have not considered $E_{\text{non-consumed}}$ in the following equations

its fraction, but a scheduler has not yet decremented $\mathrm{threads}$ before this thread has got preempted. To account this, $E_{\mathrm{frac}_{i+1}}$ must be defined as follows:

$$E_{\mathrm{frac}_{i+1}} = \frac{E_{\mathrm{frac}_i} \cdot \mathrm{threads} - E_{\mathrm{received}_{\mathrm{preempted}}}}{\mathrm{threads}_{\mathrm{receive}}} \tag{3.7}$$

Here, $\mathrm{threads}$ is the number of remaining lower priority threads permitted to receive the offered energy, and $\mathrm{threads}_{\mathrm{receive}}$ is the number of higher priority threads willing to receive the offered energy. $E_{\mathrm{received}_{\mathrm{preempted}}}$ is the already received energy of the preempted thread.

Case 2 Equation (3.6) will not apply if the preempted thread continues with its execution, because its already received energy is not considered. If one prohibits the preempted thread to receive more energy, it may receive less energy than the subsequent threads of its priority-queue. Alternatively, if one allows the preempted thread to receive more energy, and if one does not consider its already received energy, a scheduler cannot fairly partition this received energy among the eligible threads.

Therefore, if a preempted thread continues with its execution, $E_{\mathrm{frac}_{i+1}}$ has to be defined as:

$$E_{\mathrm{frac}_{i+1}} = \frac{E_{\mathrm{frac}_i} \cdot \mathrm{threads} + E_{\mathrm{received}_{\mathrm{preempted}}}}{\mathrm{threads}_{\mathrm{receive}}} \tag{3.8}$$

If $E_{\mathrm{frac}_{i+1}} < E_{\mathrm{received}_{\mathrm{preempted}}}$, the preempted thread has received more energy than permitted by the new fraction of offered energy $E_{\mathrm{frac}_{i+1}}$. Therefore, the preempted thread must not receive more energy. In addition, this equation indicates a run-queue energy budget allowing to transfer more energy than previously offered. It permits each of the remaining $\mathrm{threads}_{\mathrm{receive}} - 1$ threads to receive more energy than previously offered:

$$\frac{E_{\mathrm{received}_{\mathrm{preempted}}} - E_{\mathrm{frac}_{i+1}}}{\mathrm{threads}_{\mathrm{receive}} - 1} \tag{3.9}$$

To avoid transferring more energy than offered before, a scheduler must check whether $E_{\mathrm{frac}_{i+1}} < E_{\mathrm{receive}_{\mathrm{preempted}}}$ (3.8). If this is the case, $E_{\mathrm{frac}_{i+1}}$ must be defined as:

$$E_{\mathrm{frac}_{i+1}} = \frac{E_{\mathrm{frac}_i} \cdot \mathrm{threads}}{\mathrm{threads}_{\mathrm{receive}} - 1} \tag{3.10}$$

After the preempted thread has executed for its quantum, a scheduler only decrements $\mathrm{threads}$. Thus, the run-queue energy budget defines how much energy a scheduler can transfer.

3.4.3 Maximum Capacity of the Run-Queue Energy Budget

At last, we outline in this section why the capacity of the energy budget has to be limited and on which factors the limitation depends.

The idea of the run-queue energy budget is to permit to exhaust a processor's power limit over a hyper-period of threads' quanta. If some threads which are eligible to be scheduled offer more energy than other threads receive, the run-queue energy budget will accumulate this non-consumed offered energy. A scheduler can transfer this offered energy to threads requiring this energy for their best performance later on. Exhausting the accumulated energy in a short period of time can cause problems, e.g., the transferred energy can raise a processor's temperature beyond the threshold, or can permit threads to receive more energy than permitted per hyper-period.

Therefore, the maximum capacity of the run-queue energy budget must be defined to avoid a violation of one of these limits. We will outline how one can determine the maximum capacity in the case of limiting a processor's temperature for avoiding its exceedance above the processor's critical temperature. Therefore, we consider Kellner's temperature model of a processor [20].

Example We assume an ambient air temperature of $24.0°\mathrm{C}$, a processor idle temperature of $33.5°\mathrm{C}$ and $72.5°\mathrm{C}$ as the processor's critical temperature. The maximal power dissipation of the processor is $115W$ and the permitted average power consumption P_{limit} is $80W$. This average power consumption yields a processor temperature of about $70.1°\mathrm{C}$. To assure that a thread cannot raise the processor's energy above the critical temperature by performed energy transfers, we have to consider a processor's maximal power dissipation. The capacity of the run-queue energy budget must be exhausted before a thread consuming the maximal power can raise the processor's temperature above the threshold. In our example, a thread dissipating the maximal power will raise the processor's temperature above its critical temperature after $54s$ if the processor temperature is $70.1°\mathrm{C}$. Consequently, the maximum capacity of the run-queue energy budget may be at most

$$E_{\text{total offered}} = t_{\text{maximal runtime}} \cdot (P_{\max} - P_{\mathrm{limit}}) \tag{3.11}$$

Here, $t_{\text{maximal runtime}}$ is the maximal execution time of a thread consuming the processor's maximal power $P_{\max}$ before it raises the processor's temperature yielded by P_{limit} above the critical temperature. Considering our example, the maximum capacity of the run-queue energy budget may be at most $1,855J$.

In our example (cf. Figure 3.16), at first the processor is idle for $100s$ and yields a temperature of $35.5°\mathrm{C}$. Afterwards, a thread with an average power consumption of $80W$ executes for $2200s$ and raises the processor temperature to $70.1°\mathrm{C}$. If the thread executes also for the following $100s$, the processor temperature will remain constant, but if we replace the thread with another one dissipating $115W$, the processor temperature will exceed the threshold temperature after executing that thread for $54s$.

3.5 Controlling a Thread's Energy Limit

The last section of our design considers how a thread's energy budget can be met over several quanta. It cannot be met during one quantum, because the kernel cannot account the energy consumption constantly. Thus, a thread exceeds its energy budget before a scheduler can schedule the next thread.

Depending on the energy policy, the length of a thread's quantum can be based on a thread's energy budget. If this is the case, a scheduler must account the energy consumption of a thread periodically, to determine how long its quantum may last and when its energy budget is exhausted, respectively. A scheduler can perform this during the timer interrupt handling. As long as a thread's energy budget has not been exhausted, a scheduler executes the thread at least for the time until the next timer interrupt occurs.

If a thread has exhausted its energy budget, a scheduler will schedule the next thread. Usually, a thread has consumed more energy than permitted by the energy limit, because a scheduler accounts a thread's energy consumption only at distinct points in time. Besides, it depends on a scheduler whether it directly performs a thread switch

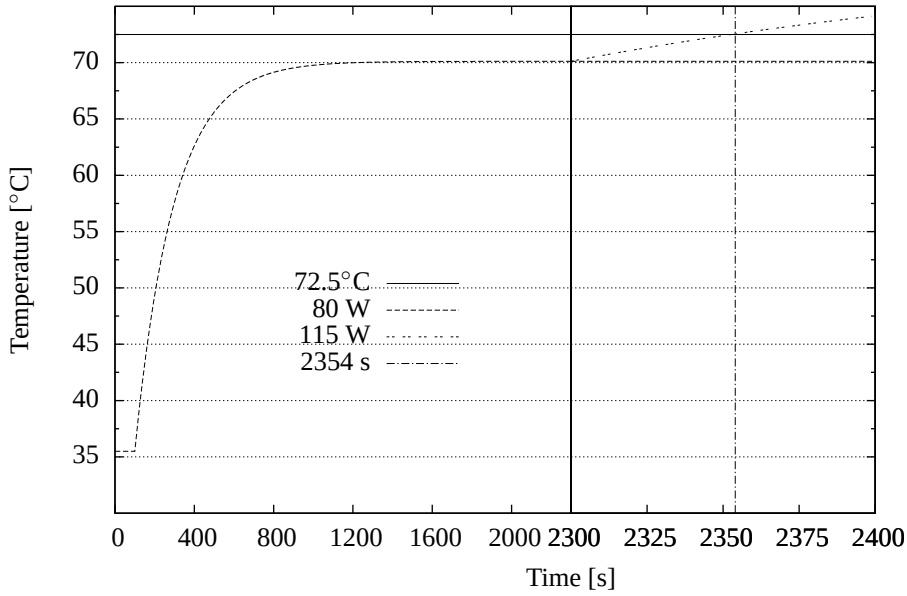

Figure 3.16: Processor Temperature

after the accounting or whether the kernel delays the thread switch. Consequently, a thread's energy consumption can significantly differ from its energy limit. In order to preserve the threads' energy budgets over several schedules, the additional, non-permitted energy must be accounted.

We call it *extra used energy* $E_{\text{extra used}}$ and it is defined as:

$$E_{\text{extra used}_i} = P_{\text{average}} \cdot t_{\text{quantum}} - E_{\text{budget}_{\text{thread}}} + E_{\text{extra used}_{i-1}} \tag{3.12}$$

To penalize a thread in its $i+1^{\text{th}}$ quantum for its extra consumed energy caused during its i^{th} quantum, a thread will lose processor control if the following equation is true:

$$E_{\text{budget}_{\text{thread}}} \leq P_{\text{average}} \cdot t_{\text{quantum}} + E_{\text{extra used}_i} \tag{3.13}$$

If a thread loses processor control because it has exhausted its quantum or has blocked, a scheduler will reset the thread's energy budget after it has calculated the thread's extra used energy. A scheduler will set $E_{\text{extra used}}$ to zero, if the last equation (3.13) is false. This can happen if a thread blocks or we apply the non-strict energy limit, causing a thread to lose processor control, although it has not exhausted its energy budget.

A scheduler must not reset a thread's energy budget if it does not reset a thread's quantum after the thread has been preempted by another thread. If a thread's energy budget was reset, it would cause to reset the thread's quantum. This would permit the thread to execute once again for the duration of its complete quantum. Thus, by steadily interrupting a thread's execution, a scheduler would reset the thread's energy budget periodically. This would possibly cause starvation of other threads.

Chapter 4

Implementation

After we have presented our design assuring a fair energy partitioning among threads, this chapter addresses implementation details of our design on top of the Linux 2.6.22.15 kernel. Our implementation is based on the former works of Waitz [40] and Merkel [25]. They have implemented the support for accounting a thread's energy consumption as well as estimating a thread's power consumption.

This chapter at first addresses how one can realize the throttling mechanism in Linux for preserving a pre-defined power limit. Afterwards, we consider when we can avoid to update a thread's energy profile to increase the accounting period for estimating a thread's power consumption per update of a thread's energy profile. Furthermore, we outline which enhancements we have performed for handling preemptions within a thread's quantum and for accounting appropriately a thread's energy transfers. We point out how to estimate the number of throttled threads or their weight and tickets, respectively. Afterwards, we propose the Linux sysfs interfaces to change the energy policy as well as the semantics of energy transfers and to define the maximum capacity of a run-queue's energy budget. At the end of this chapter, we motivate why some scheduling policies will require to be notified if a user changes the energy policy and which data structures must be updated then. At last, we consider scheduler specific adaptions to realize our proposed design.

4.1 Power Limit

In order to limit a thread's power consumption, in our design we have considered throttling as the mechanism of choice, because it allows to meet a pre-defined power limit (cf. Subsection 3.2.2). Throttling is achieved by executing a special instruction of a processor, which simply consumes a processor's idle power consumption, e.g., the `hlt` instruction of x86 processors [11]. This instruction is architecture specific, thus Linux executes an architecture specific idle thread to execute the instruction. We discuss in this section how we can throttle a thread as well as assure a proper accounting of a thread's throttling and outline when we can avoid to account a thread's power consumption.

Thread Throttling For throttling a thread in Linux, a scheduler must schedule a processor's idle thread at least for the duration of one timer tick. We have proposed to account a thread's power consumption while handling the timer interrupt, and to throttle

afterwards the current thread if necessary. Linux forbids to schedule a thread while handling the timer interrupt, therefore it is not possible to schedule the idle thread eagerly. Instead, it must be done lazily by setting the current thread's `TIF_NEED_RESCHED` flag. This flag indicates the kernel to schedule another thread. The kernel evaluates this flag before it returns to user mode and after it has handled the timer interrupt.

The flag only indicates a scheduler to schedule another thread, but not which one. Hence, a scheduler would not schedule the idle thread next, because at least the thread to be throttled is runnable. Therefore, we have extended a processor's run-queue data structure by a pointer pointing to the scheduled thread which might be throttled. The pointer `scheduled_thread` indicates which thread is executed, that is why it is unimportant whether the thread is throttled or not. In contrast to the pointer `curr`, pointing to the currently executed thread of a run-queue, a scheduler will not change the pointer `scheduled_thread` if it schedules the idle thread in order to throttle a thread. Thus, if a scheduler does not throttle a thread, the pointers `scheduled_thread` and `curr` will both point to the same thread. If a scheduler throttles a thread, `scheduled_thread` will point to the throttled thread currently not executed and `curr` to the idle thread currently executed. Besides, we have extended a thread's thread control block (TCB) by a `resched` flag. A scheduler will set the flag if it shall throttle a non-throttled thread or shall no longer throttle a throttled thread.

These two enhancements allow a scheduler to decide whether it has been called because it shall schedule another thread, throttle a thread or stop a thread's throttling. If the scheduled thread's `resched` flag is set, a scheduler will throttle or stop throttling the scheduled thread. Otherwise, a scheduler has been called to schedule another thread. Due to the lazy thread scheduling, it is possible that a scheduler has set a thread's `TIF_NEED_RESCHED` flag more than once before the kernel evaluates it. Therefore, a scheduler may only set the flag `resched` if it has not set the `TIF_NEED_RESCHED` flag before. Additionally, if a scheduler sets this flag after it has set the `resched` flag, it must reset the `resched` flag to preempt the scheduled thread.

As pointed out, for throttling a thread, a scheduler must schedule the processor's idle thread. Consequently, the idle thread is the current thread. This raises the following problems:

- A scheduler does not charge a thread for its throttling, because a scheduler does not decrease the thread's remaining timeslice length as long as the idle thread is scheduled. Moreover, a scheduler does even not increase a thread's runtime. Consequently, a thread's throttling extends its quantum.
- A thread's user and system times will not increase, if a scheduler executes the idle thread.

To solve these problems, we replace the pointer to the current thread with the pointer to the scheduled thread, whenever it is necessary.

Power Consumption Accounting If a thread does not exceed the power limit, it can be an advantage to avoid accounting a thread's power consumption on each timer tick. Therefore, we have to consider under which circumstances we can avoid the accounting within the timer interrupt.

To estimate the number of threads which might receive energy if we allow energy transfers, we proposed in Subsubsection 3.4.2.1 to account how many threads have been or would have been throttled. Therefore, we have extended a thread's TCB with two flags: `throttled` and `throttled_last`. These two flags indicate whether a

scheduler has throttled or has intended to throttle a thread during the thread's current or previous quantum. The latter flag is not required for estimating how many threads may receive energy, instead its purpose is to extend the period while a scheduler accounts a thread's power consumption on each timer tick.

Thus, a scheduler will not require to account a thread's power consumption on each timer tick if both flags are not set. Nonetheless, it is required to account a thread's power consumption periodically after k timer ticks to conclude whether the thread's power consumption is still below the power limit.

4.2 Energy Profile

We outline in this section when a scheduler has to update the complete data structures of a thread's energy profile, or when it is sufficient to update only a subset of it, to account a thread's power consumption and its energy consumption.

In the following three subsections, we explain how we can avoid to update a thread's energy profile each time a thread's power or energy consumption is accounted. We want to decrease the frequency of updating a thread's energy profile in order to increase the accounting period for estimating a thread's power consumption per update. Therefore, we extend the energy profile by the field `consumed_energy_accounting_period`. It stores the energy consumed during a thread's quantum. In addition, we point out how to update a preempted thread's and its preemptor's energy profile for accounting their consumed energy accurately. To account their consumed energy, we extend the energy profile by the field `preemption_energy`. At last, we discuss how the energy transfer can be realized as presented in our design. It requires to extend the energy profile by the field `received_energy`. This accounts the amount of energy a thread has received due to energy transfers.

4.2.1 Updating a Thread's Energy Profile

A thread's energy profile contains a thread's amount of total consumed energy and a thread's exponential average energy consumption. This exponential average smoothes the change of a thread's energy consumption. The former works solely required to update a thread's energy profile before scheduling the next thread. Thus, the kernel can save the performance counter values as well as a thread's consumed energy in appropriate data structures after accounting them. Our proposed policies require to account a thread's energy consumption – depending on the energy policy – on every timer tick for estimating at least a thread's power consumption or even for limiting a thread's energy consumption during its quantum. Our proposed policies, however, do not require to update the data structures each time the power or energy consumption is accounted.

By updating these values after each timer tick, a scheduler decreases the period for accounting a thread's current power consumption. The period is diminished to the duration of one timer tick; without updating the values it can last up to a thread's quantum. If a scheduler updates the power consumption on each timer tick, it will weight the power consumption caused during each period of one timer tick exponentially. Otherwise, it will only weight the average power consumption consumed during a thread's quantum exponentially. Consequently, the thread's current power consumption has a greater impact on its estimated power consumption in the latter case. Aside from the

longer accounting period, this mechanism reduces the overhead for maintaining the energy profile, but this is only a minor reason.

Not only the power limit requires to account a thread's consumed energy on each timer tick, the energy limit does require this as well. To account a thread's power consumption, the accounting mechanism simply needs to yield the exponentially weighted power consumption. Limiting a thread's energy consumption requires to account its consumption during its quantum, additionally. Therefore, a scheduler must update the sum of energy a thread has already consumed. Otherwise, it must save how much energy a thread has consumed since the last update of a thread's energy profile. We have chosen the latter approach and have therefore extended a thread's energy profile by the field `consumed_energy_accounting_period`. A scheduler is required merely to account each time a thread's power or energy consumption; it does not need to update the complete energy profile. It updates only the field `consumed_energy_accounting_period` and the energy profile yields the estimated power consumption.

4.2.2 Handling Preemptions

In Section 3.5 of our design, we have pointed out that a scheduler must not reset a thread's energy budget if it does not reset a thread's quantum after a thread gets preempted, otherwise we would not account a thread's energy budget appropriately. Contrary to the former works, we must account the total consumed energy caused during a thread's quantum. Thereby, it is unimportant whether a thread gets preempted during its quantum or not.

The field `consumed_energy_accounting_period` accounts the energy consumed since the last complete update of its energy profile's data structure. In the case of priority based schedulers, we update a thread's energy profile not only after the thread has exhausted its energy budget, but also after a thread has been preempted. Thus, the field `consumed_energy_accounting_period` will no longer reflect a thread's consumed energy within its quantum if a scheduler reads it after a thread gets preempted.

To account how much energy the thread has already consumed within its quantum, we must enhance the thread's energy profile. A scheduler increases the field `preemption_energy` by the energy a thread has consumed during its last scheduling period when it gets preempted. Furthermore, a scheduler must reset the `consumed_energy_accounting_period` to zero, thus `preemption_energy` and `consumed_energy_accounting_period` together mirror a thread's consumed energy during its quantum. A scheduler must reset the `preemption_energy` to zero after a thread has blocked or exhausted its quantum. Hence, it is possible to account a thread's energy consumption appropriately, even in the case of priority based schedulers.

4.2.3 Energy Transfer

The extended energy profile simply allows to control a thread's energy limit as well as its power limit, but it is not possible to realize the energy transfer, as proposed in Subsection 3.4.2. In order to perform the energy transfer, we must account how much offered energy a thread has already received, to assure that it does not receive more offered energy than permitted by E_{frac}. To permit energy transfers, we extend the energy profile by the field `received_energy`. After a scheduler has accounted a thread's energy consumption within the timer tick, it calculates the energy a thread

may have received. It updates the received energy as described in Appendix (A.3). Due to the performed energy transfers, a thread's power consumption can be beyond the allowed limit. Because we do not measure a thread's power consumption, but estimate the power consumption – it is based on a thread's previous energy consumptions –, a scheduler would throttle a thread due to an average power consumption above the limit if the power limit was not reset to the allowed one.

Only reseting a thread's power consumption to the pre-defined power limit does not avoid its throttling, because we have chosen to update a thread's energy profile only after the thread has lost processor ownership. Consequently, a scheduler considers a thread's power consumption caused during the performed energy transfers for the estimated power consumption. To avoid this, a scheduler must update a thread's energy profile after a thread has exceeded its assigned fraction of transferred energy. After reseting the power consumption to the pre-defined limit, the thread behaves as expected from the perspective of the power limit. To account a thread's energy budget appropriately, the energy consumed during its schedule must be considered until the update of its energy profile. Analogously to the case of preempting a thread, a scheduler increases `preemption_energy` by `consumed_energy_accounting_period` and resets the latter value to zero. Only in the case of priority based schedulers the preemption energy can be non-zero. Otherwise, a scheduler uses this value only to permit a proper accounting of a thread's energy consumption. Thus, it is possible to account a thread's power and energy consumption appropriately.

4.3 Counting Throttled Threads

We have considered in our design to partition the offered energy only among threads being eligible to be scheduled and probably willing to receive this energy. This section addresses how one can account these threads.

In the case of non-priority based schedulers, it is sufficient to account the total number of throttled threads. If a scheduler applies a proportional share scheduling policy, it must account a throttled thread's number of tickets and its weight, respectively, instead of accounting the thread itself.

We will consider a thread as willing to receive offered energy if a scheduler has or would have throttled a thread within the thread's current or previous quantum. To avoid accounting the number of probably receiving threads, their weights or number of tickets $\mathrm{threads}_{\mathrm{receive}}$ every time before partitioning the offered energy, we store a counter per run-queue. For a correct accounting, it is necessary to know whether a thread has already increased the counter and must not increase the counter another time, or whether it has already increased the counter and may now decrease it. Besides, we have extended a thread's TCB with the `receive_energy` flag. This flag indicates whether a thread has already increased the counter or not.

A scheduler will increase the counter and will set a thread's `receive_energy` flag if

- a thread gets enqueued into the run-queue and a scheduler has throttled the thread within the thread's current or previous quantum before the thread got dequeued, or
- a thread's `receive_energy` flag is not set and a scheduler has throttled the thread within the thread's currently expired quantum.

Opposite to incrementing the counter, a scheduler will decrement the counter and will reset a thread's `receive_energy` flag to zero if this flag is set and

- a thread gets dequeued from the run-queue, or
- a scheduler has not throttled the thread within the thread's currently expired quantum and the thread is still enqueued.

In the case of proportional share scheduling policies, a scheduler must update the counter according to a thread's number of tickets or weight. Additionally, if a user changes a thread's weight or number of tickets and a thread's `receive_energy` flag is set, a scheduler must update the counter appropriately.

Priority based schedulers require one counter per priority-queue. A runnable thread's priority change requires to dequeue the thread from its current priority-queue and enqueue it into its new priority-queue, hence a scheduler can update the counters analogously to the run-queue's counter of a non-priority based scheduler.

4.4 Sysfs Interface

This section deals with the Linux sysfs interface to select one of our four proposed energy policies. Besides, it allows to set the maximum capacity of a run-queue energy budget and to choose which energy transfer semantics shall be applied.

Normally, a user cannot change the internal data structures of the Linux kernel. Nonetheless, Linux offers two interfaces to export internal kernel data structures from kernel space to user space: the proc filesystem and sysfs filesystem [7]. We have chosen the sysfs interface to allow the user to select one of the proposed energy policies, to set the maximum capacity of a run-queue's energy budget, and to choose the semantics of energy transfers. The proc filesystem is used to define the power limit as described in [25].

4.4.1 Energy Policy

To allow the user to select one of our four proposed energy policies on runtime, we have extended the virtual sysfs filesystem with the file `/sys/kernel/energy_policy`. A user can select one of the four following policies by writing the policy's number into the file. When reading this file, it provides the names and descriptions of the four policies and indicates which of these policies is active (cf. Table 4.4.1).

Number	Policy
0	strict power & non-strict energy limit
1	strict power & strict energy limit
2	non-strict power & non-strict energy limit
3	non-strict power & strict energy limit

Table 4.1: Energy Policies

4.4.2 Maximum Capacity of a Run-Queue's Energy Budget

To avoid a violation of a processor's critical temperature or a higher energy consumption during a long period of time, it is required to limit the maximum capacity of a run-queue's energy budget. A user can set the limit by writing the new maximum capacity measured in μJ into the file `/sys/kernel/maximum_capacity_energy_budget`. When reading the file, it returns the current maximum capacity of a run-queue's energy budget in μJ.

We would set the maximum capacity of the run-queue's energy budget of our example discussed in Subsection 3.4.3 to $1,855,000,000 \mu J$.

4.4.3 Energy Transfer

In our design, we have outlined two different semantics for an energy transfer: the transferred energy can be given away or can merely permit a thread to exceed the power limit.

Therefore, we have added the file `/sys/kernel/give_energy_away` to the virtual sysfs filesystem. Reading this file supplies the user with the information which semantics a scheduler currently applies for energy transfers. It will return zero if the energy transfer only permits to exceed the power limit, and it will return one if the transferred energy is given away by the offering threads.

4.5 Changing the Energy Policy

The sysfs file `/sys/kernel/energy_policy` allows the user to specify which energy policy a scheduler shall apply. In this section, we outline which scheduling policies will require to update their own internal data structures, and the TCB structure of the system's threads if a user changes the applied energy policy.

This applies especially to the SFQ scheduling policy outlined in Subsection 2.2.3. If we select the strict power limit and the non-strict energy limit energy policy and do no longer apply the strict energy limit energy policy, the scheduler must reset its virtual time and maximum finish tag as well as each thread's start and finish tag. The virtual time and these tags have been based on a thread's consumed energy before. After the switch, they are based on a thread's exhausted fraction of its timeslice measured in timer ticks. A thread's consumed energy causes the virtual time to increase much faster than the exhausted fraction of a thread's timeslice due to the finer resolution. Therefore, the resulting gap between the virtual times of the individual threads is substantially larger in the first case than in the latter. If one did not reset the virtual time and the tags, the thread with the lowest virtual time would execute for a long period of time, until its virtual time would be larger than any other thread's virtual time of the run-queue. Consequently, it is necessary to reset the scheduler's and threads' virtual time related fields.

To avoid the overhead of accounting the throttled threads for energy policies implementing the strict power limit, each scheduler resets its counter for the throttled threads as well as the `receive_energy` flag of each thread. Therefore, the kernel notifies the current scheduler when the user changes the energy policy. It does not notify the current scheduler when the user changes the semantics of the energy transfers, because the semantics simply defines whether a thread's received energy influences the length of a thread's quantum or not, but does not affect data structures.

4.6 Scheduler Specific Adaptions

In addition to our proposed enhancements to permit a fair energy partitioning, some schedulers will require changes for adapting their specific characteristics and applying our proposed design if the scheduling decision is based on a thread's energy consumption. This applies to the CFS and the $O(1)$ scheduler. In the following two subsections, we outline the adaptions for these two schedulers.

4.6.1 $O(1)$ Linux Scheduler

The Linux scheduler does not only favor interactive threads by increasing their priority in contrast to decreasing the priority of CPU-bound threads. Additionally, it splits the timeslices of interactive threads into several small pieces (cf. Subsection 2.2.6). The scheduler reinserts a thread into the run-queue after the thread has executed for the duration of the piece of its timeslice. This increases the reactivity of interactive threads, thus it shall also be possible if the scheduling decision is based on a thread's energy consumption.

At first, we outline how the original timeslice based scheduler implements this mechanism. Afterwards, we propose our adaption for the energy based energy policies. The scheduler will reinsert an interactive thread into the run-queue if the thread resides in the active array of the run-queue and its remaining timeslice is not smaller than its timeslice granularity. A thread's timeslice granularity assures that the scheduler executes a thread for a minimum number of timer ticks before it may reinsert the thread. Therefore, the following equation must be fulfilled in addition:

$$0 \equiv (\mathrm{timeslice}_{\mathrm{total}} - \mathrm{timeslice}_{\mathrm{remaining}}) \bmod \mathrm{timeslice}_{\mathrm{granularity}} \tag{4.1}$$

Here, $\mathrm{timeslice}_{\mathrm{total}}$ is the timeslice length of the priority-queue a thread is assigned to, $\mathrm{timeslice}_{\mathrm{remaining}}$ the remaining length of its timeslice and $\mathrm{timeslice}_{\mathrm{granularity}}$ the previously introduced timeslice granularity of a thread's priority. If an interactive thread fulfills these three conditions, the scheduler will reinsert the thread into the thread's priority-queue.

In the case of an energy based scheduler, a thread's consumed energy does not simply increase by one unit. This applies only to a thread's remaining timeslice decreasing after each timer tick by one unit. Therefore, it is inadequate to solely adapt the previous equation (4.1) for splitting a thread's energy budget, because its fulfillment is very improbable. This can be seen in the following equation:

$$0 \equiv E_{\mathrm{budget}_{\mathrm{thread}}} \bmod (P_{\mathrm{limit}} \cdot \mathrm{timeslice}_{\mathrm{granularity}}) \tag{4.2}$$

Instead, we have to allow for a non-predictable increase of a thread's consumed energy from one timer tick to another. The scheduler will split an interactive thread's energy budget if the thread is enqueued in the active array, and the thread's energy consumption fulfills the following inequation:

$$E_{\mathrm{consumed}} \geq (\mathrm{timeslice}_{\mathrm{split}} + 1) \cdot (P_{\mathrm{limit}} \cdot \mathrm{timeslice}_{\mathrm{granularity}}) \tag{4.3}$$

E_{consumed} is the energy a thread has already consumed during its quantum, and $\mathrm{timeslice}_{\mathrm{split}}$ counts how often the inequation has already been fulfilled within the quantum. Thereto, on each timer tick the scheduler checks whether the thread's consumed energy solves this inequation. If this is the case, the scheduler will increment

$\mathrm{timeslice}_{\mathrm{split}}$. Afterwards, if the thread is interactive, the scheduler has just incremented $\mathrm{timeslice}_{\mathrm{split}}$ and the thread is enqueued into the active array, the scheduler will reinsert the thread into the thread's priority-queue. In this way, it is possible to split a thread's energy budget into small fragments, analogously to splitting a thread's timeslice.

4.6.2 Completely Fair Scheduler

For our implementation we have used the Linux 2.6.22.15 kernel. This kernel implements the presented $O(1)$ scheduler. From Linux kernel version 2.6.23 on, the $O(1)$ scheduler has been replaced by the Completely Fair Scheduler (cf. Subsection 2.2.7). To evaluate this new Linux scheduler, we have applied a patch replacing the old scheduler with the CFS of the Linux 2.6.24.1 kernel. Molnar [26] has provided the patch.

The CFS uses timeslices of variable length which it dynamically determines to realize the proportional share among the threads. Therefore, a thread's weight extends a thread's timeslice length, but does not affect how often the scheduler schedules a thread, like in the case of the other presented proportional share scheduling policies. If merely one thread is runnable, the thread's timeslice length will be unbound, until another thread will become runnable. Therefore, CFS avoids to check whether another thread is eligible to be scheduled or not.

As outlined in Section 4.2, we only want to update a thread's energy profile before scheduling the next thread, after a thread has completely received the energy E_{frac} or has exhausted its energy budget. If just one thread is runnable, the scheduler will never update a thread's energy profile. In this case, the thread's estimated power consumption is equal to its average power consumption caused since its schedule. Therefore, the scheduler must update a thread's energy profile after the period of k timer ticks, if it has not updated the energy profile within this period.

In addition to an unbound timeslice of a thread, the CFS has – in contrast to the remaining proportional share scheduling policies – a two-level hierarchy for its proportional share and not only one. The top-level defines the proportional share between different groups or users, whereas the bottom-level defines the proportional share between the threads assigned to a group or to a user. An administrator can group a user's threads to an administrator defined group together. We use in the following the more abstract concept of groups for our considerations, but they apply for users as well. To account the weight of all throttled threads, it is insufficient to account only their own weight, instead the scheduler must consider the weight of the group a thread is assigned to as well. The scheduler represents a thread's group by a per group runqueue, called `CFS_rq`. Each group is assigned to the top-level `CFS_rq`. We have extended the data structure by a counter. This counter accounts the weight of the throttled threads assigned to the `CFS_rq`. If the scheduler considers a thread to be willing to receive energy, the scheduler will increment at first the counter of the thread's group by the thread's own weight $\mathrm{thread}_{\mathrm{weight}}$. Afterwards, the scheduler will increment the counter of the top-level `CFS_rq` by $\mathrm{thread}_{\mathrm{weight}} \cdot \mathrm{group}_{\mathrm{weight}}$.

Thus, if a user changes a group's share, the CFS can update the new total weight of the throttled threads easily. The scheduler must update the top-level counter as follows:

$$\mathrm{counter}_{\mathrm{top}} = \mathrm{counter}_{\mathrm{top}} + (-\mathrm{group}_{\mathrm{weight}_{\mathrm{old}}} + \mathrm{group}_{\mathrm{weight}_{\mathrm{new}}}) \cdot \mathrm{counter}_{\mathrm{group}} \quad (4.4)$$

In this way, it is possible to efficiently account the weight of the throttled threads, even in the case of a multi-level proportional share scheduling policy.

Chapter 5

Evaluation

This chapter deals with the evaluation of our proposed design. At first, we describe our evaluation environment, and compare the performance of our seven examined schedulers with each other. Subsequently, we evaluate whether each of these schedulers can fulfill the demands of our designed energy policies. Afterwards, we examine whether the proportional share scheduling policies performing our designed energy transfer can fairly partition the offered energy according to the threads' weights. Furthermore, we discuss the implications of I/O load on our fair energy partitioning. At last, we evaluate the performance of our designed energy transfer.

5.1 Evaluation Environment

We evaluate the performance and fairness of our implementation on a simultaneously multithreaded $3.8GHz$ Pentium 4 processor with two logical processors and $2GByte$ memory. To preclude an influence of a load balancing mechanism for our evaluation, we have deactivated one of the two logical processors. Due to timeslice lengths of $10ms$ in case of proportional share schedulers, we have set the timer frequency to $1000Hz$.

To analyze our design we have chosen two benchmarks of the SPEC CPU2006 benchmark suite [35]: the `hmmer` benchmark and the `lbm` benchmark (cf. Table 5.1). We have chosen these two benchmarks, because the `hmmer` benchmark can benefit from the `lbm` benchmark if we permit energy transfers. Furthermore, the `hmmer` benchmark requires more energy for its execution, but has a shorter runtime than the `lbm` bench-

Bench[a]	**Min PC [W]**[b]	**Av PC [W]**[c]	**Max PC [W]**[d]	**Av RT [s]**[e]	**EC [J]**[f]
hmmer	106	107	110	1,209	128,700 J
lbm	93	94	96	1,343	127,000 J

[a] Benchmark
[b] Minimum Power Consumption
[c] Average Power Consumption
[d] Maximum Power Consumption
[e] Average Runtime
[f] Energy Consumption

Table 5.1: Benchmarks

mark. This permits to explore the consequences of our energy policies. For our considerations, we only consider the energy consumption of the CPU, but not of the complete system.

To investigate the interrupt latency induced by our enhancements, and to measure the network performance, we have attached an Intel E1000 Gigabit network interface card to the system. An external client and our target system execute the `Netperf` benchmark [27] to generate I/O load on our system. The `Netperf` benchmark has an average power consumption of approximately $100W$.

5.2 Evaluation Setup

To evaluate our generic design, we have set the average power limit of our system to $100W$ and restricted the maximum capacity of a run-queue's energy budget to $300J$. We have chosen $300J$ to permit the `hmmer` benchmark to exceed the power limit about $10W$ for $30s$. In order to only compare the performance of our seven examined schedulers, we have not applied these restrictions and deactivated energy accounting.

We have set the timeslice lengths of the schedulers to the following values:

Scheduler	**Timeslice Length [*ms*]**
$O(1)$ Scheduler	100
MLFQ Scheduler	5,7,10,14, ..., 824
Round Robin (RR) Scheduler	100
SFQ Scheduler	10
Stride Scheduler	10
Lottery Scheduler	10
CFS	variable

Table 5.2: Timeslice Length

The timeslice length of 100*ms* is the default timeslice length of the $O(1)$ scheduler. We have chosen this timeslice length for the round robin scheduler as well, and set the timeslice length of SFQ, stride and lottery schedulers to 10*ms* to favor I/O-bound threads. To assign the appropriate timeslice length to each thread, the MLFQ scheduler has a specific timeslice length per priority. The CFS does not have a concept of timeslices. It determines a thread's time of processor control dynamically.

We have proposed four different energy policies and two semantics for energy transfers. Therefore, we have to evaluate six different energy policies. In addition, we evaluate the performance of a per run-queue throttling policy to examine whether we can achieve a better performance by throttling each thread individually than by throttling the complete run-queue. The run-queue throttling policy throttles the current thread as long as the estimated power consumption is beyond the power limit. Therefore, `hmmer`'s power consumption and exceedance of the power limit can cause `lbm`'s throttling, resulting in an unfairness among the two benchmarks. The abbreviations of our policies are shown in Table 5.3.

The kernel does not display its output on a monitor, but sends it through the serial interface. This permits to log the kernel messages from an external client. To evaluate a thread's energy consumption, we have extended the exit system call. It transfers to the serial interface the data how often a thread was throttled, not throttled, or its

Energy Policy	Energy Transfer	Abbrev.
Strict Power & Non-Strict Energy Limit	None	SPNE
Strict Power & Strict Energy Limit	None	SPSE
Non-Strict Power & Non-Strict Energy Limit	Extended E. Budget	NPNE1
Non-Strict Power & Non-Strict Energy Limit	Exceeded P. Limit	NPNE2
Non-Strict Power & Strict Energy Limit	Extended E. Budget	NPSE1
Non-Strict Power & Strict Energy Limit	Exceeded P. Limit	NPSE2
Run-Queue Throttling	None	RQTH

E. = Energy
P. = Power
Abbrev. = Abbreviation

Table 5.3: Energy Policy Abbreviations

throttling was avoided due to energy transfers. Additionally, it transmits a thread's runtime, consumed energy and received energy.

5.3 Scheduler Performance

Before we evaluate the fairness and performance of our design, we compare the performance of the seven examined schedulers. Therefore, we have executed the complete SPEC CPU2006 int and fp benchmark suite per scheduler. We have executed each individual benchmark of the benchmark suites three times. To evaluate the overhead caused by an individual scheduler in comparison to our reference scheduler, the $O(1)$ scheduler, we have selected the shortest runtime of each benchmark out of the three runs. The overall execution times of the SPEC CPU2006 int and fp benchmark suite are shown in Table 5.4 and in Table 5.5, respectively.

The overhead in the following two evaluations indicates the overhead for executing the complete benchmark suite by a specific scheduler compared to our reference scheduler. Minimum and maximum overhead outline how far the performance between the investigated and the reference scheduler drifts throughout a benchmark suite.

Scheduler	Runtime [s]	Ovh [%][a]	Min Ovh [%][b]	Max Ovh [%][c]
$O(1)$ Scheduler	12511.94	0.00	0.00	0.00
MLFQ Scheduler	12522.01	0.08	-0.6	0.93
RR Scheduler	12550.15	0.31	-1.08	1.21
SFQ Scheduler	12573.78	0.49	-0.64	1.84
Stride Scheduler	12551.92	0.32	-0.62	1.51
Lottery Scheduler	12571.02	0.47	-0.46	2.19
CFS	12518.46	0.05	-0.19	2.44

[a] Overhead
[b] Minimum Overhead
[c] Maximum Overhead

Table 5.4: SPEC CPU2006 int Benchmark Suite

Scheduler	Runtime [s]	Ovh [%]	Min Ovh [%]	Max Ovh [%]
$O(1)$ Scheduler	23459.74	0.00	0.00	0.00
MLFQ Scheduler	23508.65	0.21	-0.71	0.94
RR Scheduler	23553.68	0.40	-0.09	3.08
SFQ Scheduler	23544.92	0.36	-0.11	1.43
Stride Scheduler	23530.82	0.30	-0.70	4.03
Lottery Scheduler	23512.79	0.23	-0.75	3.45
CFS	23500.38	0.17	-0.67	1.5

Table 5.5: SPEC CPU2006 fp Benchmark Suite

For an individual benchmark, the runtime difference between the reference scheduler and one of the six remaining schedulers is at most about 4%. The overall overhead for executing the benchmark suite by one of the non-reference schedulers is at most 0.5% in comparison to the reference scheduler. Therefore, we expect that the later evaluation results are comparable among the seven schedulers.

5.4 Energy Policies

In Subsection 3.3.4, we have discussed the expected performance of our proposed energy policies. To expose the effect of energy transfers, we have selected the `hmmer` benchmark and the `lbm` benchmark. We execute these benchmarks in parallel and set the power limit to $100W$. If a scheduler performs energy transfers, the `hmmer` benchmark can benefit from the `lbm` benchmark's lower power consumption. Independently of performed energy transfers, we expect that a scheduler cannot completely avoid the `hmmer` benchmark's throttling.

5.4.1 Scheduler Comparison

Before we examine the impacts of the six energy policies and the run-queue throttling policy on these two benchmarks individually, we analyze the impacts of the energy policies on the total execution time of this scenario. It lasts from the start of these two benchmarks until both have finished with their execution.

Energy Policies The non-strict power limit reduces the total execution time of this scenario significantly as seen in Figure 5.1. Depending on the scheduler, the total execution time can be reduced between 3.1% and 4.8% in comparison to the two policies realizing the strict power limit.

If a scheduler's energy policy forbids energy transfers, `hmmer` must spend between 8.3% and 9.0% of its executed ticks for its throttling. Otherwise, if we permit energy transfers, a scheduler will throttle `hmmer` at most about 3.5%. Depending on the estimated power consumption of `lbm` – it varies between $93.3W$ and $96.4W$ –, a scheduler can avoid completely `hmmer`'s throttling. This applies to SFQ and stride schedulers as well as CFS while applying one of the following energy policies: NPSE1, NPNE2 or NPSE2. If a scheduler applies one of these energy policies, `lbm` has a longer lasting quantum in comparison to `hmmer`. Due to its longer execution time per quantum, it can

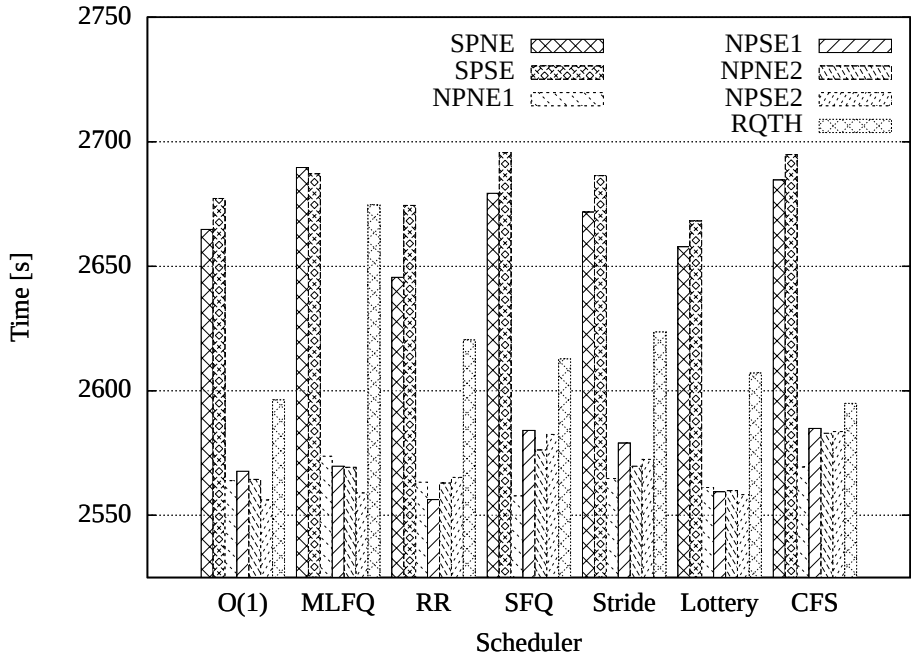

Figure 5.1: Scheduler Comparison - Total Execution Time

offer more energy than in the case of applying the NPNE1 energy policy and therefore avoiding `hmmer`'s throttling. Applying the strict energy limit in contrast to the non-strict energy limit and permitting energy transfers reduces the total execution time in the case of $O(1)$, MLFQ, round robin and lottery schedulers. This outlines that a fair energy partitioning does not necessarily increase the total execution time of several threads.

The lottery scheduler is a probabilistic proportional share scheduling policy. Consequently, the scheduling order of these two benchmarks is nondeterministic. The scheduler can schedule a benchmark consecutively for several quanta and not only for one which reduces the number of context switches. If the scheduler schedules the `hmmer` benchmark consecutively for several quanta, `hmmer` can consume up to the complete offered energy of the `lbm` benchmark and enforces its own throttling. The `hmmer` benchmark must spend at most 1.4% of its executed ticks for its own throttling if the scheduler permits energy transfers. Nevertheless, due to the reduced number of context switches, the performance of the lottery scheduler is comparable to the performance of the remaining schedulers. Among our seven examined schedulers, the total execution time of the two benchmarks differs at most between 0.6% and 1.7% for the six energy policies. Consequently, the aim of each energy policy can be achieved independently from a specific scheduler. To execute the two benchmarks our energy-aware schedulers with the exception of the CFS scheduler require approximately 3.3% more time than an unmodified Linux $O(1)$ scheduler. The CFS requires 3.7% more time than an unmodified Linux $O(1)$ scheduler, but only 2.8% more time than an unmodified CFS.

Run-Queue Throttling Policy After we have compared the energy policies with one another, we compare them with the run-queue throttling policy. The run-queue throttling policy has been applied in the former works of Waitz [40] and Merkel [25] to limit a processor's power consumption and temperature, but not to limit them per thread like in our work. To limit a processor's temperature, a run-queue throttling policy is sufficient, but it induces unfairness because it can throttle threads which have not consumed more power than permitted.

If we prohibit energy transfers, the run-queue throttling policy will achieve a better performance than our strict power limit energy policies, because the run-queue throttling policy preserves a processor's power limit over a hyper-period of several thread's quanta. Therefore, it will finish the execution of both benchmarks between 0.5% and 3.3% earlier. In comparison to the fastest energy policy of each scheduler permitting energy transfers, the run-queue throttling policy requires between 1.0% and 4.5% more time to finish the execution of this scenario.

5.4.2 Comparison of Energy Policies

After we have compared the performance of the schedulers with one another and have come to the conclusion that each scheduler can fulfill the aims of the energy policies, we analyze the implications of the six energy policies on the `hmmer` benchmark and the `lbm` benchmark. We have chosen the $O(1)$ scheduler to evaluate the performance of these benchmarks. The performance of the remaining schedulers is comparable. In Figure 5.2, we show the runtime of the benchmarks for each energy policy.

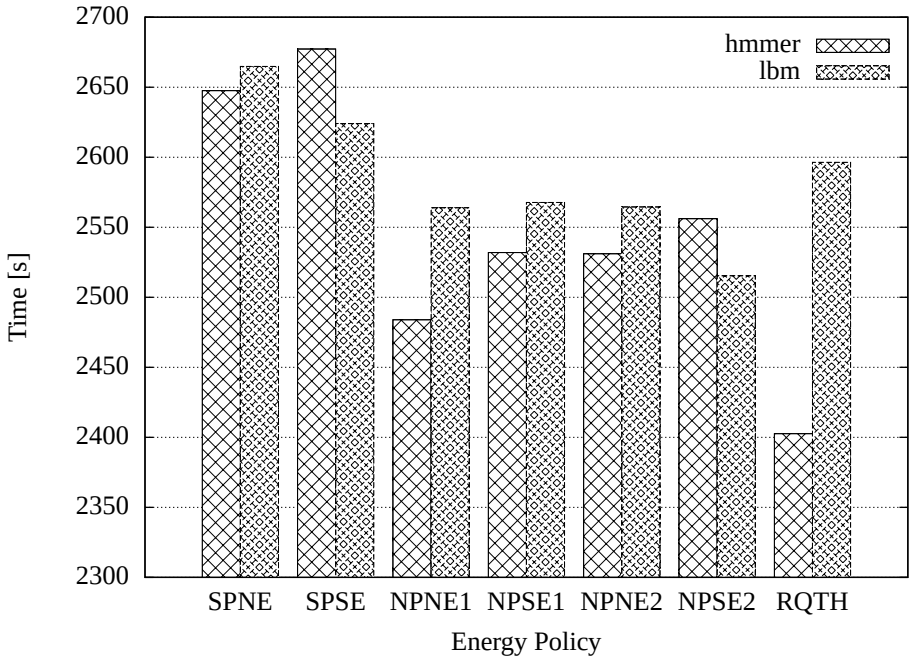

Figure 5.2: Energy Policy Comparison

Energy Policies The hmmer benchmark cannot benefit from the lower power consumption of the lbm benchmark if we apply energy policies enforcing the strict power limit. In case of the strict power limit and the non-strict energy limit energy policy, hmmer finishes its execution earlier than lbm, because its runtime is shorter than lbm's runtime. This energy policy does not consider that hmmer consumes more energy.

In comparison to that energy policy, the strict power limit and the strict energy limit energy policy favors lbm by extending its quantum due to its low power consumption. Therefore, lbm finishes its execution first, because in total it consumes less energy during its execution than hmmer. In this particular case, hmmer requires $53s$ longer for its execution than lbm. This time corresponds roughly to the greater amount of energy ($5124J$) it requires.

As pointed out in Subsection 3.3.3, a thread receiving offered energy will benefit at most from the non-strict power limit and the non-strict energy limit energy policy if the offered energy extends the thread's energy budget. This applies to the hmmer benchmark as well. It finishes its execution $80s$ before lbm finishes its execution. We have analyzed in Subsection 3.3.4 that threads offering energy do not benefit from their offered energy. Nevertheless, lbm has finished $101s$ earlier than in the case of the prohibited energy transfers. This results from the fact that the lbm benchmark has to compete for processor control only for $2484s$ and not for $2648s$.

The gap of $80s$ between the two benchmarks can be diminished to $36s$ if the scheduler applies the strict energy limit in contrast to the non-strict energy limit. The scheduler cannot achieve a fair energy partitioning, because the lbm benchmark gives its offered energy away. If the lbm benchmark permits the hmmer benchmark to exceed its power limit but not to extend its energy budget, the gap between the two benchmarks can be reduced significantly.

Yet, the hmmer benchmark will still finish $33s$ before the lbm benchmark if we apply the non-strict energy limit. In contrast to that, the lbm benchmark will finish $41s$ earlier its execution than the hmmer benchmark if the offered energy is not given away. In general, if a scheduler applies the strict energy limit in contrast to the non-strict energy limit, the gap between these two benchmarks can be at least diminished by about 45% or the lbm benchmark can finish even earlier than the hmmer benchmark.

In the case of the deterministic proportional share scheduling policies, we cannot recognize a significant difference between the NPNE2 and NPSE2 energy policies. These schedulers realize a proportional share based on the threads energy consumptions. Therefore, lbm can execute for another time if lbm's virtual time and energy, respectively, are below hmmer's one.

Run-Queue Throttling Policy As outlined before, the run-queue throttling policy requires more time for the execution of the two benchmarks than most of our energy policies. This results from throttling the lbm benchmark, which has a longer runtime than the hmmer benchmark. Thus, the drawback of the run-queue throttling policy is that it can also throttle threads which have not exceeded the power limit. In Table 5.6, we outline how often a scheduler has throttled hmmer and lbm during their execution.

The drawback of the run-queue throttling policy becomes most obvious in the case of the MLFQ scheduler. It causes hmmer to be throttled in 0.1% of its executed ticks and lbm to be throttled in 4.0% of its executed ticks by the scheduler. The scheduler's throttling of lbm further increases lbm's runtime. This benchmark finishes its execution $294s$ later than hmmer. In the case of our energy policies, lbm finishes its execution at most $165s$ later than hmmer.

Scheduler	H. Th. Ticks	H. To. Ticks	L. Th. Ticks	L. To. Ticks
$O(1)$ Scheduler	2700	1189484	43488	1392513
MLFQ Scheduler	1383	1194582	59553	1477219
RR Scheduler	2700	1187132	1000	1419862
SFQ Scheduler	410	1093335	600	1396696
Stride Scheduler	170	1091977	80	1391428
Lottery Scheduler	1470	1086519	1580	1398468
CFS	1475	1213850	2521	1379748

H. = Hmmer
L. = Lbm
Th. = Throttled
To. = Total

Table 5.6: Run-Queue Throttling

5.5 Proportional Share Schedulers

In contrast to round robin or MLFQ schedulers, a proportional share scheduler permits to fairly partition the time of processor allocation or the processor energy among threads according to their weights. We want to evaluate how fairly a scheduler partitions the offered energy among threads with different weights.

Therefore, we have executed two `hmmer` benchmarks and one `lbm` benchmark in parallel. The `lbm` benchmark and one of the `hmmer` benchmarks (`hmmer-1`) have weight one and the second `hmmer` benchmark has weight two (`hmmer-2`). The `hmmer-2` benchmark has to execute two iterations of the benchmark, instead of one iteration like the two other benchmarks. Thus, the two `hmmer` benchmarks shall finish at the same time.

In contrast to the previous scenario, the offered energy of the `lbm` benchmark is insufficient to avoid a throttling of the two `hmmer` benchmarks, because we have set the power limit to $100W$. Nonetheless, energy transfers can significantly reduce the total execution time of this scenario as shown in Figure 5.3. Among our four proportional share scheduling policies, the total execution time of the three benchmarks differs at most between 0.2% and 2.1% for the six energy policies. Due to the throttling of the two `hmmer` benchmarks, our energy-aware schedulers require at most 2.0% more time for the execution of the three benchmarks than an unmodified CFS.

The run-queue throttling policy requires between 1.1% and 2.6% more time for the execution of this scenario than our energy policies permitting energy transfers. This applies to each of the four schedulers. Like in the previous scenario, the run-queue throttling policy will finish the execution of this scenario earlier if we apply the strict power limit.

After we have analyzed the total execution times, we exemplarily analyze the runtimes of the individual benchmarks for the stride scheduler. The results of the three remaining schedulers are comparable for the six energy policies and the run-queue throttling policy.

Energy Policies If we apply the extended energy budget energy transfer, the two `hmmer` benchmarks will finish earlier than `lbm`. The `lbm` benchmark will finish earlier its execution than the two `hmmer` benchmarks if we apply the exceeded power limit

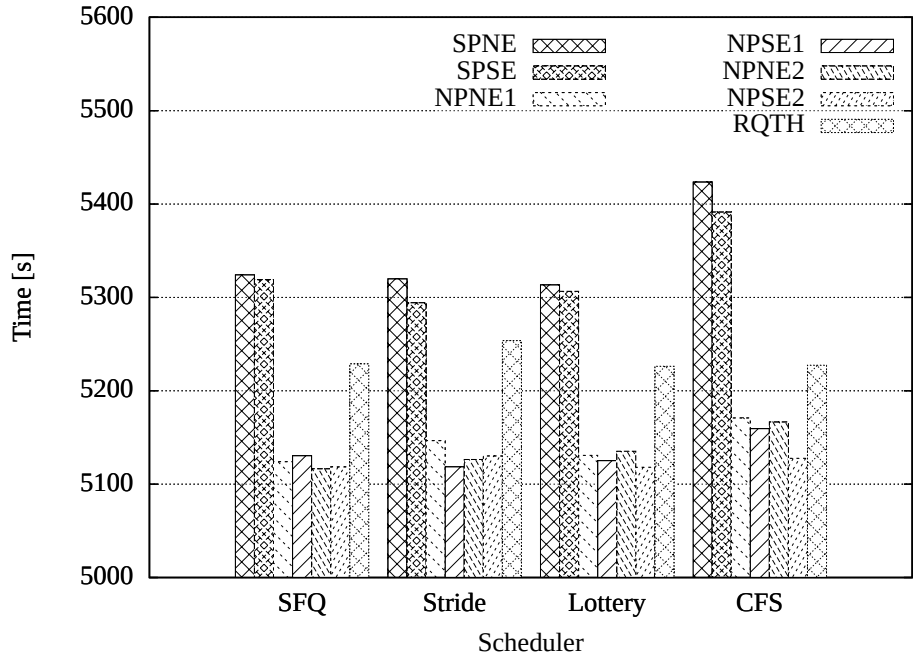

Figure 5.3: Proportional Share Scheduler Comparison - Total Execution Time

energy transfer. It benefits from a fair energy partitioning. If we apply the NPSE2 energy policy, it will finish $76s$ before `hmmer-2` and $95s$ before `hmmer-1`. The gap between the `hmmer` benchmarks and the `lbm` benchmark will be larger, if we apply the strict power limit and the strict energy limit energy policy. In this case, the `lbm` benchmark will finish $159s$ before `hmmer-2` and $170s$ before `hmmer-1`.

Run-Queue Throttling Policy As outlined before, our seven schedulers can achieve a better performance if they apply our energy policies permitting energy transfers in contrast to applying the run-queue throttling policy. The run-queue throttling policy causes a performance degradation, because it throttles the `lbm` benchmark. This benchmark has to spend between 4.4% and 9.6% of its executed ticks for its throttling. Therefore, `lbm` finishes its execution between $165s$ and $308s$ after the slower one of the two `hmmer` benchmarks has finished its execution.

Energy Policies At last, we compare the two `hmmer` benchmarks with each other. To examine whether a fixed timeslice length is a drawback in comparison to a timeslice of variable length, we examine the performance of the stride scheduler and the CFS. For comparing the two `hmmer` benchmarks with each other, we consider the following ratios: runtime, consumed energy (energy sum), received energy, throttled ticks, not throttled ticks and avoided throttled ticks due to energy transfers. The run-queue throttling policy does not transfer energy, therefore we have set the ratios received energy and avoided throttling to zero. We outline the first three ratios in Figure 5.4 and the latter three ratios in Figure 5.5.

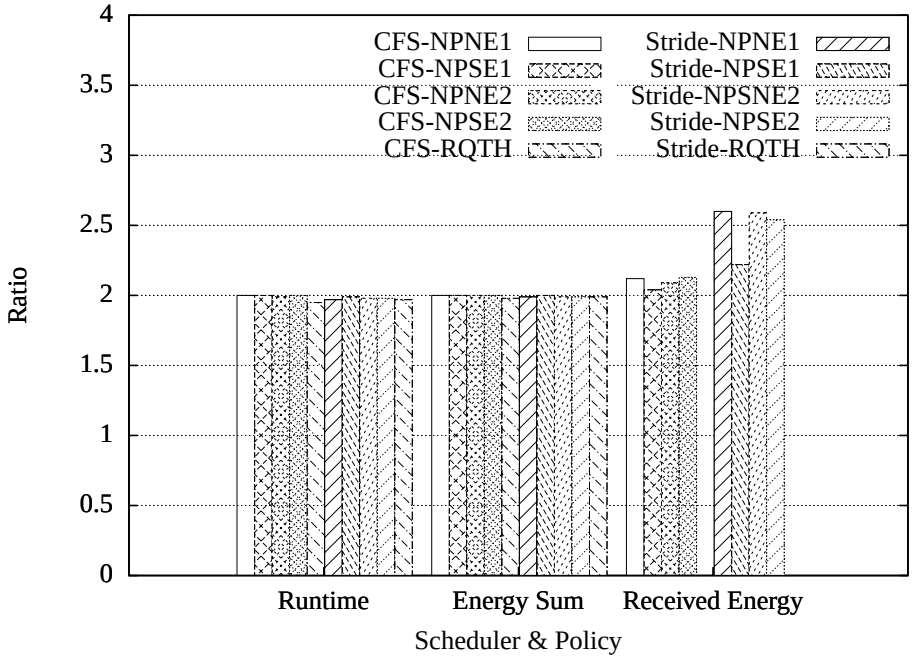

Figure 5.4: Proportional Share Scheduler Comparison - Hmmer Ratios Part 1

The ratios runtime and amount of consumed energy correspond to the ratio of the two `hmmer` benchmarks' weights, independently from the applied energy policy or the scheduler. In the case of the stride scheduler the ratio received energy does not correspond to the ratio of the two `hmmer` benchmarks. The `hmmer-2` benchmark receives more than twice the energy of the `hmmer-1` benchmark. Therefore, `hmmer-2` will finish $43s$, $13s$, $18s$ and $19s$ earlier than the other benchmark if we apply NPNE1, NPSE1, NPSE1, NPNE2 and NPSE2 energy policies, respectively. The $43s$, $13s$, $18s$ and $19s$ correspond to runtime ratios of 1.97, 1.99, 1.98 and 1.98, respectively. These ratios result from the fixed timeslice length of the stride scheduler. To achieve a proportional share among the threads, it schedules a thread having k times the weight of another thread k times more often. The thread with a higher weight has a greater chance to receive the offered energy. This, however, does not apply to the CFS, because this scheduler uses timeslices of variable length and therefore does not schedule a thread more often than another one. Consequently, each thread receives its designated offered energy.

Considering the ratios throttled, not throttled and avoided throttling, one can see that these ratios vary significantly. If the stride scheduler throttles the `hmmer-2` benchmark less than twice as often as the `hmmer-1` benchmark, `hmmer-2` will receive a bigger share of offered energy than expected. This does not affect the ratios runtime and energy sum significantly, because the `hmmer` benchmarks must spend at most 5.3% of their timer ticks for their throttling. Consequently, our proposed energy transfer in combination with our energy policies achieve to partition the energy among threads fairly according to their weights.

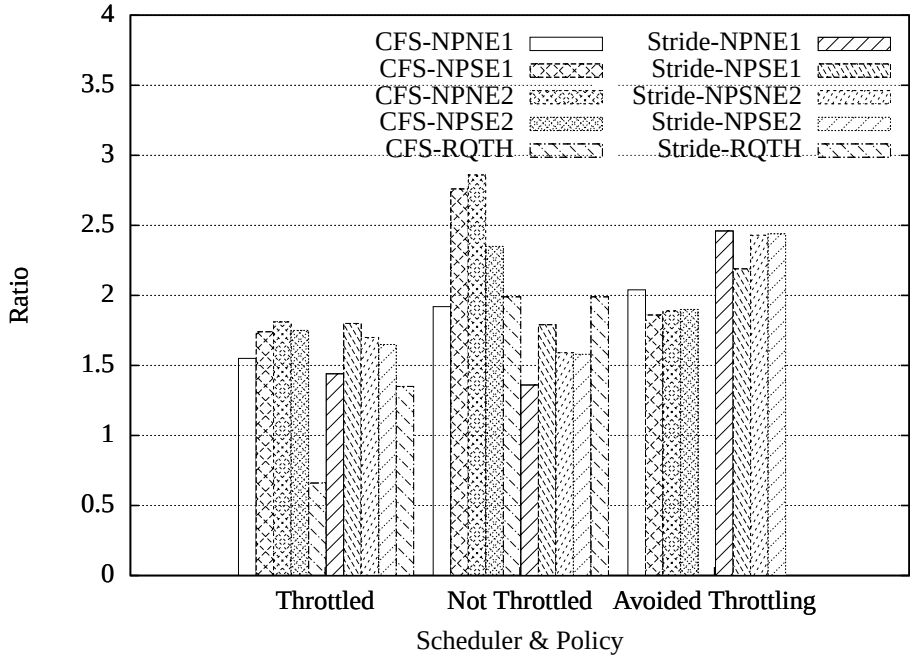

Figure 5.5: Proportional Share Scheduler Comparison - Hmmer Ratios Part 2

Run-Queue Throttling Policy If we apply the run-queue throttling policy, the gap between the two `hmmer` benchmarks will be $19s$, $40s$, $3s$ and $72s$ when the benchmarks will be executed by SFQ, stride, lottery and completely fair schedulers, respectively. This gap will result from throttling `hmmer-2` not twice as often as `hmmer-1`. In the case of the CFS, the gap between `hmmer-2` and `hmmer-1` is $72s$, because CFS executes `hmmer-2` twice as long as `hmmer-1`, but not twice as often as the remaining schedulers. Hence, `hmmer-2` can benefit twice as long from `lbm`'s throttling and lower power consumption if `hmmer-2` is scheduled right after `lbm`. Therefore, `hmmer-1` is throttled 1.5 times more often than `hmmer-2`. The other schedulers throttle `hmmer-2` between 1.4 and 2.4 times more often than `hmmer-1`.

5.6 Interactive Tasks

In the previous sections, we have evaluated the performance of CPU-bound tasks, but have not considered I/O-bound tasks. To analyze how an I/O-bound thread affects our proposed fair energy partitioning, we execute the `Netperf` benchmark [27] as well as `hmmer` and `lbm` in parallel. We set once again the power limit to $100W$. The `Netperf` benchmark sends as many data packages as possible from an external client to our system within $10s$. Afterwards, it returns the average throughput achieved within this time. The benchmark executes over and over again until the `hmmer` benchmark and the `lbm` benchmark have finished with their execution. An unmodified Linux system can achieve a throughput of approximately $852\frac{Mbit}{s}$. We analyze in the following two

subsections `Netperf`'s throughput and the implications of the generated I/O load on the `hmmer` benchmark and the `lbm` benchmark.

5.6.1 Netperf Throughput

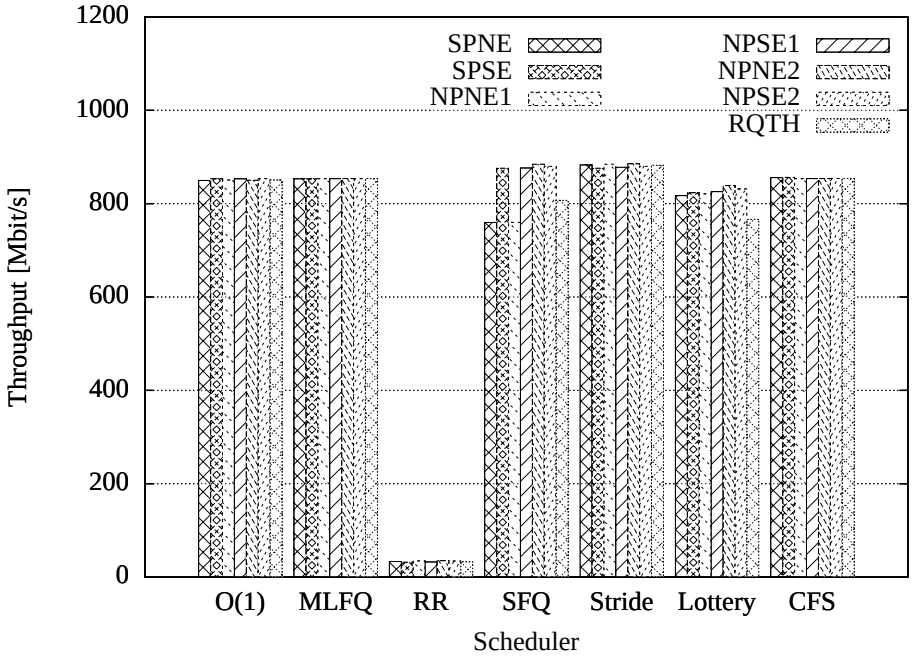

Figure 5.6: Scheduler Comparison - Netperf Performance

CFS as well as $O(1)$ and MLFQ schedulers achieve the throughput of the unmodified $O(1)$ Linux scheduler of about $852\frac{Mbit}{s}$. Stride and SFQ schedulers achieve the highest throughput of about $880\frac{Mbit}{s}$, whereas the round robin scheduler achieves the lowest throughput of about $35\frac{Mbit}{s}$ as shown in Figure 5.6. With the exception of the SFQ scheduler, these six schedulers have in common that the variation of throughput does not depend on the applied energy policy. Merely the throughput achieved by lottery and SFQ schedulers varies substantially.

SFQ and lottery schedulers will achieve a throughput of approximately $759\frac{Mbit}{s}$ and $825\frac{Mbit}{s}$, respectively, if the scheduling decision is based on the thread's execution time (SPNE and NPNE1) and of about $880\frac{Mbit}{s}$ and $830\frac{Mbit}{s}$, respectively, if the scheduling decision is based on a thread's energy consumption (SPSE, NPSE1, NPNE2 and NPSE2). To benefit I/O-bound threads, the SFQ scheduler increases a thread's virtual time only by the fraction of a thread's timeslice the thread has executed for. If the SFQ scheduler measures a thread's period of execution in coarse-grained timer ticks, it will overestimate a thread's execution time. An overestimated execution time causes that a thread's virtual time increases further than necessary. This will be not the case, if the SFQ scheduler measures a thread's execution time in nanoseconds. Then, the scheduler schedules `Netperf` more frequently, which increases `Netperf`'s throughput.

Similar to the SFQ scheduler, the lottery scheduler benefits from measuring a thread's execution time in nanoseconds and not in timer ticks, because it has to determine the fraction of a thread's timeslice a thread has executed for. The lottery scheduler requires this fraction f, because it scales a thread's tickets with the factor $\frac{1}{f}$ to increase an I/O-bound thread's chance to be elected the next time the thread becomes runnable.

The CFS and our implementation of the stride scheduler measure a thread's execution in nanoseconds, therefore they do not suffer from the problem of an overestimation of a thread's execution time. This outlines that measuring a thread's execution time in timer ticks is to coarse-grained for proportional share policies.

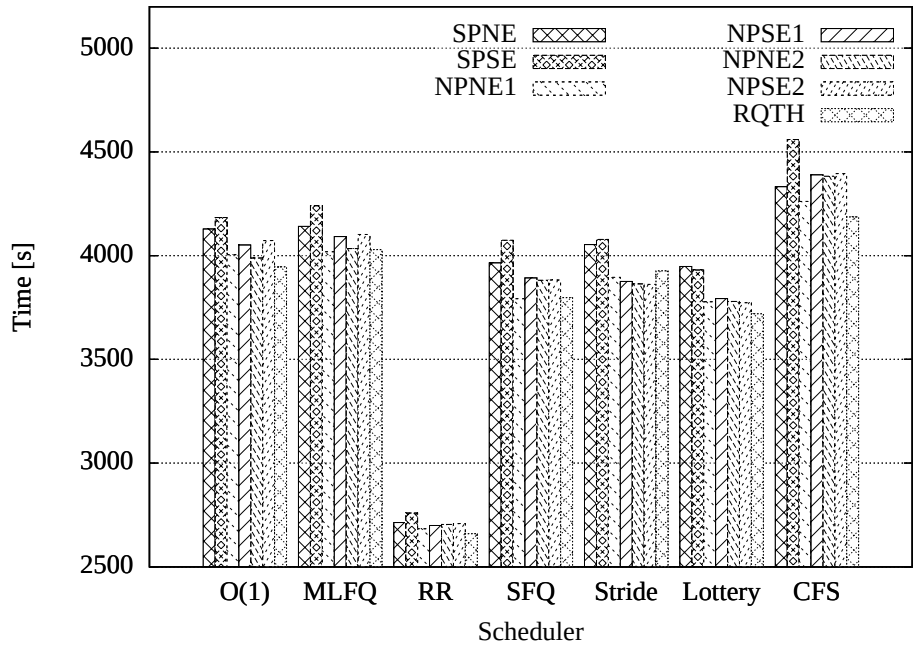

Figure 5.7: Scheduler Comparison - Total Execution Time of Hmmer and Lbm While Executing Netperf

Although the CFS is not a priority based scheduler, it will favor I/O-bound threads by inserting them at the front of the scheduling timeline if a thread has slept just for a short period of time. Due to the high rate of interrupts caused by the `Netperf` benchmark, the benchmark sleeps only for a short period of time and therefore preempts the currently executed thread to handle the outstanding interrupts. In the case of $O(1)$ and MLFQ schedulers, the `Netperf` benchmark has a higher priority than `hmmer` and `lbm` and thus preempts these ones.

Contrary, SFQ, stride, round robin and lottery schedulers do not preempt the current thread and therefore delay the interrupt handling. How long the delay lasts, depends on the number of threads in the run-queue and the scheduler's timeslice length. The timeslice length of $100ms$ of the round robin scheduler delays the interrupt handling considerably. The achieved throughput is simply about $35\frac{Mbit}{s}$.

Although SFQ, stride and lottery schedulers all have a timeslice length of $10ms$, the throughput gap between the lottery scheduler (below $840\frac{Mbit}{s}$) on the one hand and the stride and SFQ schedulers ($880\frac{Mbit}{s}$) on the other hand is noticeable. The gap results from lotteries passing until the scheduler elects the `Netperf` benchmark to be scheduled. SFQ and stride schedulers permit the `Netperf` benchmark to achieve a better performance than $852\frac{Mbit}{s}$, because `Netperf` can handle more outstanding interrupts when they execute it, which reduces the overhead. In case of SFQ, stride and lottery schedulers, `Netperf`'s throughput will decrease if `Netperf` has to compete with more than the two benchmarks for processor control, because it is scheduled less frequently and therefore loses interrupts. As pointed out in Subsection 5.4.2, one cannot recognize a significant performance difference between the NPNE2 and NPSE2 energy policies. Therefore, the throughput for the two policies is comparable.

To evaluate the overhead caused by the logic of our energy policies and by the energy accounting mechanism, stride and SFQ schedulers are inappropriate. They delay the interrupt handling and thereby achieve the highest throughput among the schedulers. Therefore, we think that CFS as well as $O(1)$ and MLFQ schedulers are better indicators for the overhead. We cannot notice a throughput loss, but an increase of the runtimes of `hmmer` and `lbm`. For the execution of both benchmarks, $O(1)$ and MLFQ schedulers require 2.4% and 3.2% more time, respectively. The overhead of the CFS is more significant. It requires 9.4% and 7.1% more time for the execution of this scenario than the unmodified $O(1)$ scheduler and CFS, respectively.

5.6.2 Benchmark Runtime

After we have stated that the throughput of the `Netperf` benchmark depends on the scheduler and its timeslice length, but not on the energy policy, we analyze the implications of the generated I/O load on the `hmmer` benchmark and the `lbm` benchmark.

Energy Policies Energy transfers can decrease with one exception the total execution time of this scenario at least about 0.3% and at most about 5.3% as shown in Figure 5.7. If the CFS applies the NPNE1 energy policy, it will increase the total execution time of this scenario about 1.2% in comparison to applying the SPNE energy policy. In Section 5.1, we have pointed out that the runtime of the `lbm` benchmark is longer than the runtime of the `hmmer` benchmark. Hence, the generated I/O load affects the `lbm` benchmark more seriously than the `hmmer` benchmark, because `lbm`'s execution is interrupted more often, and in the case of schedulers favoring I/O-bound threads preempted more often. Although preempting the current thread assures a quicker handling of the outstanding interrupts, it degrades the throughput and increases the total execution time of the two benchmarks. The throughput degradation and increase of the total execution time result from the additionally required address space switches and from estimating the threads' energy consumptions.

Due to these drawbacks, it depends on the scheduler whether the `lbm` benchmark finishes earlier than the `hmmer` benchmark in the case of SPSE or NPSE2 energy policies. Nevertheless, all schedulers have in common that they can reduce the gap between `hmmer` and `lbm` if we apply the strict energy limit. Thus, even in the case of I/O-bound threads, our strict energy limit energy policies favor the `lbm` benchmark as intended.

Run-Queue Throttling Policy The performance of the run-queue throttling policy is comparable to the performance of the NPSE1 energy policy. This applies to the

individual runtimes of the two benchmarks as well to the total execution time of both benchmarks. The runtimes vary at most about 1.8%.

5.7 Evaluation of Energy Transfer

At last, we evaluate the performance of our examined schedulers applying the non-strict energy limit and the strict energy limit. These two policies do not limit a processor's power consumption. We want to ascertain whether our energy policies permitting energy transfers and preserving a power limit of $100W$ can achieve – due to the energy transfers – the performance of the non-strict energy limit and the strict energy limit not applying a power limit.

The non-strict energy limit preempts a thread at the latest after the thread has executed for the period of its timeslice, the strict energy limit executes a thread until the thread has consumed the energy of its energy budget. To evaluate the performance of the schedulers applying the strict energy limit, we have set the power limit to $100W$. This power limit does not cause a thread's throttling, it is only required to define a thread's energy budget (3.1). We compare the results of the schedulers applying the non-strict energy limit and the strict energy limit with the results achieved by the schedulers applying our energy policies preserving the power limit of $100W$. Our energy policies prohibiting energy transfers and therefore strictly preserving the power limit cannot achieve the performance of the non-strict energy limit and the strict energy limits, because they must throttle `hmmer` to preserve the power limit. This applies to each of the previously discussed scenarios.

Energy Policies In our first scenario, we have only executed the `lbm` benchmark and the `hmmer` benchmark in parallel. If we apply the non-strict energy limit, most of the examined schedulers applying our non-strict energy limit energy policies will require between 0.2% and 1.5% more time to execute both benchmarks. The CFS and the stride scheduler can decrease the total execution time about 0.8% and 0.4%, respectively. If we apply the strict energy limit energy policies, the schedulers can reduce the execution time between 0.9% and 3.3%. Depending on the scheduler, our energy policies can decrease `lbm`'s runtime at most about 3.3%, or increase `lbm`'s runtime at most about 1.5% in comparison to the schedulers applying non-strict or strict energy limits. In contrast to `lbm`, the non-strict energy limit energy policies cannot decrease `hmmer`'s runtime. NPNE1 energy policies increase the runtime between 0.1% and 3.8%, whereas NPNE2 energy policies increase the runtime between 4.1% and 6.7%. The overhead caused by the NPNE2 energy policies results from their shorter quanta to favor the `lbm` benchmark. Our strict energy limit energy policies can decrease the runtime of `hmmer` between 0.9% and 6.1%.

Proportional Share Schedulers We have executed two `hmmer` benchmarks and one `lbm` benchmark in parallel in our second scenario. One `hmmer` benchmark has weight two (`hmmer-2`), the other two benchmarks have weight one (`hmmer-1` and `lbm-1`). Each of the latter benchmarks has to execute one iteration of its benchmark, whereas `hmmer-2` has to execute two iterations of its benchmark.

Our non-strict energy limit energy policies permitting energy transfers require between 1.0% and 3.1% more time to finish the execution of these three benchmarks. The overhead results from throttling the two `hmmer` benchmarks, it cannot be completely prevented by the offered energy of the `lbm-1` benchmark. In the case of our

non-strict energy limit energy policies, the overhead varies between 2.0% and 2.6%. In Figure 5.8, we outline the runtime of the three benchmarks for each scheduler. At first, each scheduler has applied the non-strict energy limit which preempts each thread after exhausting its timeslice. Afterwards, each scheduler has applied the strict energy limit which executes a thread until it has exhausted its energy budget. One can see that each scheduler fulfills the demands of the strict energy limit, and that the strict energy limit can mostly reduce the total execution time of this scenario.

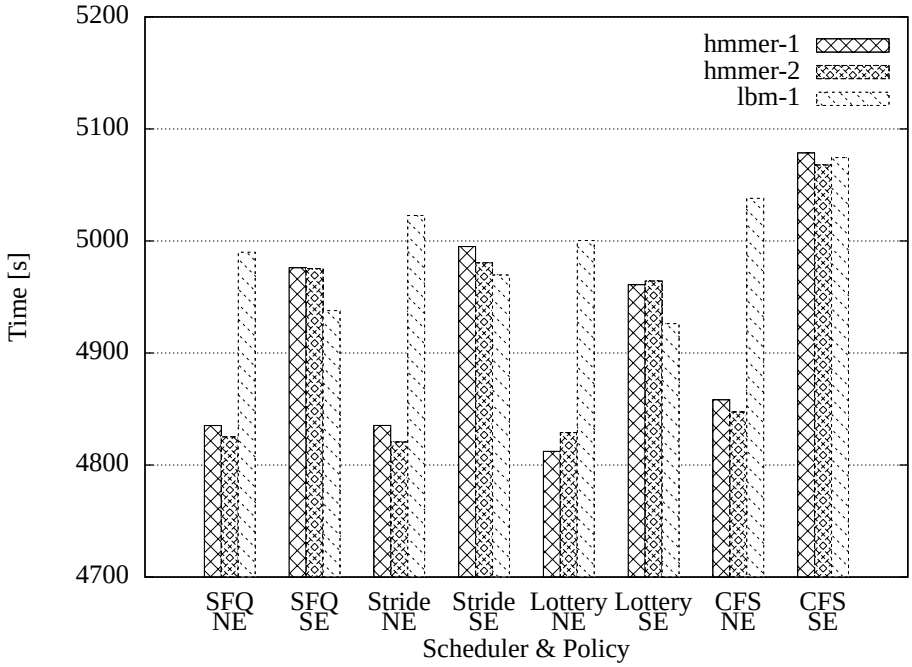

Figure 5.8: Proportional Share Scheduler Comparison - Execution Time Non-Strict Energy Limit & Strict Energy Limit

Interactive Tasks In our third scenario, we have executed `Netperf`, `hmmer` and `lbm` benchmarks in parallel. $O(1)$ and MLFQ schedulers applying our non-strict energy limit energy policies can reduce the total execution time of `hmmer` and `lbm` about 0.3% and 0.1%. The remaining schedulers cause an overhead between 2.0% and 4.6%. With the exception of $O(1)$ and MLFQ schedulers, the schedulers will increase the total execution time of the benchmarks at most about 3.9% if we apply the strict energy limit energy policies. $O(1)$ and MLFQ schedulers applying our strict energy limit energy policies reduce `lbm`'s execution time about 0.6% and 0.7%, respectively, and hence also the total execution time about 0.6% and 0.7%, respectively. As discussed previously, the throughput of `Netperf` cannot be significantly affected by our energy policies, instead it depends on a specific scheduler. This also applies to the case of not throttling our benchmarks. Therefore, the `Netperf` results are comparable to the results of our energy policies and not discussed further.

Chapter 6

Related Work

In this chapter, we present two approaches to account the energy consumption of activities in a client-server environment appropriately, and to realize a fair energy partitioning among activities. These approaches are similar to our energy policies realizing the strict power limit combined with the strict energy limit as well as the non-strict power limit combined with the non-strict energy limit.

6.1 Energy Containers

An energy container is an abstract kernel entity, permitting to account and to limit the energy consumption caused by a specific activity [6]. It is based on the concept of resource containers [4].

A resource container permits to account the usage of resources by a thread's activity in a client-server environment. Thereto, a resource container – initially bound to its creator – can be bound to several threads, and one thread can be bound to several resource containers. Thus, a thread's protection domain is no longer the limitation to account an activity's resource usage. It is possible to realize a proportional share among resource containers, because each resource container has a share defining the fraction of resource allocations it gets from its parent. The parent-child relationship among resource containers defines a hierarchy among the resource containers.

Each energy container has an assigned amount of energy. Energy containers can consume this energy within an epoch. An epoch is a period of time after which a scheduler refreshes an energy container [47]. By limiting the energy consumption of the root energy container, an exceedance of a processor's temperature above a threshold can be prevented, because the child containers can at least consume the energy of the root container. Bellosa et al. propose to penalize threads having high power consumptions and to favor threads having low power consumptions by adjusting the timeslice length of a thread. A scheduler assigns shorter timeslices to threads having high power consumptions and assigns longer timeslices to threads having low power consumptions. This is very similar to our approach of the strict power limit combined with the strict energy limit.

The major difference between the approach of energy containers and our approach is that if all energy containers have run out of energy, the processor will stop the energy containers' activities and will enter a low power state to reduce its power dissipation [6]. The energy consumed by the processor's low power state is not assigned

to the energy containers which have average power consumptions beyond the power limit and have caused the processor's throttling. Thus, a scheduler does not consider the contribution of throttling an energy container to an energy container's energy consumption. We instead do not exceed a thread's energy budget, by throttling a thread as soon as the thread's estimated power consumption exceeds the power limit, and by assigning the energy caused during a thread's throttling to the thread itself. Hence, the approach of energy containers forbids an accurate and fair energy partitioning.

6.2 ECOSystem

ECOSystem is a system [47] to manage energy as a first class resource. Like the energy containers it is based on the concept of resource containers. The resource containers, however, are not based on energy, instead they are based on a *currentcy* model to account the energy consumption of an activity. The term currentcy is a combination of the terms current and currency.

In contrast to assigning an amount of energy to an energy container, the currentcy model assigns a number of currentcy units to a resource container. A currentcy unit corresponds to a defined amount of energy which a resource at most must consume within a pre-defined time limit. If a thread does not consume all of its currentcy units, it can accumulate them. Nevertheless, the ECOSystem restricts the accumulation to avoid high peaks of power consumptions in the future. This approach is similar to our non-strict power limit and non-strict energy limit energy policy, but instead of accumulating the energy per thread, we will accumulate it per run-queue if we permit energy transfers.

If a thread runs out of currentcy before the kernel refreshes the resource container, the kernel will stop the thread's execution and will enter a low power state. Thus, an accurate and fair energy partitioning is impossible like in the case of the energy containers. They divide the time into energy-epochs as well. After each energy-epoch, a kernel thread distributes the total currentcy among the resource containers according to the shares of the resource containers. By defining the total currentcy which the system can distribute per energy-epoch, the system's average power consumption can be regulated.

The main difference between the ECOSystem and the concept of energy containers is the handling of threads having power consumptions below the power limit. ECOSystem permits to accumulate the non-consumed currentcy, whereas the energy containers extend the timeslices of these threads.

Because the ECOSystem merely limits the amount of currentcy and energy, respectively, which a thread can accumulate, but not all threads together can accumulate, it cannot prevent that threads raise a processor's temperature above a threshold. This can happen if threads consume their accumulated energy together in a short period of time.

Chapter 7

Conclusion

At first, we outline in this chapter our achievements. Afterwards, we give a short summary of our thesis, and point out possible directions of future work.

7.1 Achievements

In this thesis, we have proposed a generic design to enhance general purpose schedulers to become fair energy-aware schedulers. These schedulers can fairly partition the system's energy as well as preserve a pre-defined power limit. Each thread preserves the power limit individually, to preclude drawbacks for other threads caused by its power consumption. In contrast to the former works in the field of fair energy partitioning [6,47], we consider the energy consumed during a thread's throttling, to assure that a thread does not consume more energy as permitted in a period of time. This permits data centers to base their accounting not only on the time of processor control, but also on the caused energy consumption. The main contribution of this work is the design of a fair energy transfer to exhaust a processor's power limit over the period of several threads' quanta. It allows threads having power consumptions beyond the power limit to benefit from threads having power consumptions below the limit, while assuring a fair energy partitioning. We have achieved that our energy-aware schedulers can fairly partition the transferred energy among the receiving threads according to each thread's weight. Our energy policies applying the strict energy limit are capable of favoring energy efficient threads by extending their period of execution in contrast to energy inefficient threads. Thereby, it is unimportant whether the threads are CPU-bound or I/O-bound. By throttling each thread individually and permitting energy transfers, we achieve a better performance than by throttling the complete run-queue of a processor.

7.2 Summary

General purpose schedulers are only responsible for fairly partitioning the processor control among the threads. They do not consider the resource utilization caused by a thread during its schedule, although the resource utilization can also affect other threads. For a fair partitioning of processor assignment they should consider it.

We have proposed in this thesis a generic design to enhance schedulers to take a thread's energy consumption into account for their scheduling decisions to become

fair energy-aware schedulers. These enhanced schedulers offer four different energy policies. All four energy policies have in common that they can limit a processor's power consumption in order to avoid exceeding a processor's power limit or raising its temperature above a threshold.

Out of these four policies, two realize a fair energy partitioning by executing a thread until it has consumed its assigned energy. The two other policies preempt a thread at the latest after it has executed for the time of its timeslice to increase the system's reactivity and to discard the remaining energy assigned to a thread.

Furthermore, two of our four proposed policies – one realizing a fair energy partitioning and the other one not – request of each thread not to raise its average power consumption beyond the pre-defined power limit, which prohibits to accumulate energy. The two remaining energy policies will permit threads to exceed the power limit if other threads have average power consumptions below the limit. This permits to transfer energy among threads for exhausting a processor's power limit. Due to the energy transfers, a scheduler does not preserve the power limit over a thread's quantum, but over a hyper-period of several threads' quanta. The non-consumed energy cannot only be consumed later on by the thread itself, but also by other threads. Our per run-queue energy budget prevents that threads can cause an exceedance of a processor's temperature threshold. To assure that threads with the same characteristics receive the same amount of energy, the scheduler fairly partitions the offered energy before the threads willing to receive the energy may receive it.

We have shown that the fairness of the individual energy policies does not depend on a specific scheduler. To evaluate the fairness of the different energy policies, we have executed two benchmarks in parallel, while one benchmark has an average power consumption above the power limit and the other benchmark has an average power consumption beneath the power limit. If the energy policies permit the first benchmark to benefit from the latter's power consumption, the two benchmarks will finish earlier with their execution, because they exhaust the processor's power limit. Otherwise, the processor's average power consumption is below the pre-defined power limit. Moreover, our evaluation has shown that an instance of a thread with k times the weight of another instance, also receives k times the amount of offered energy as requested by our fair energy transfer. In summary, each scheduler can achieve a fair energy partitioning, thereby it is unimportant whether a scheduler permits or prohibits energy transfers.

7.3 Future Work

Although we have designed a per run-queue fair energy partitioning, we have evaluated our design only on a uni-processor system to preclude side-effects from the load balancing mechanism. The side-effects of different migration strategies of a load balancing mechanism should be considered in order to design a multi-processor capable fair energy-aware scheduler. Furthermore, more appropriate dynamic thermal management mechanisms like throttling should be considered for reducing a thread's power consumption. Especially, quickly changing power consumptions caused by throttling a processor only for one timer tick should be avoided in order not only to be an energy-aware scheduler for servers, but also for battery based systems. Our proposed throttling mechanism is not applicable for battery based systems, because a non-constant discharge rate reduces the capacity of a battery substantially [24]. To increase the energy saving or to offer more energy to threads willing to receive it, a scheduler should select each thread's appropriate voltage and corresponding frequency [29, 30, 44]. The

increased overhead caused by the voltage switches should be acceptable for today's processors [19].

Although preserving the power limit per thread is adequate for systems without client-server interactions, preserving the power limit per resource [4] or energy container [6] can be an advantage in client-server environments. Data centers, in particular, can benefit from accounting the energy consumption caused by an activity, by permitting energy transfers between different activities to exhaust the pre-defined power limit.

Appendix A

Power Limit

A throttling mechanism assures that the average power consumption of a thread and of a processor is not raised above a pre-defined power limit. As outlined in our design, a scheduler will strictly preserve the power limit if it forbids energy transfers. Thus, each thread has to preserve the power limit during its quantum, and threads having power consumptions above the power limit cannot benefit from threads having power consumptions beneath the limit. If we loose the requirement of preserving the power limit over the period of a thread's quantum, and it must be merely met over a hyper-period of several threads' quanta, a scheduler can transfer energy.

In the following two sections of this chapter, we discuss strict and non-strict power limits in detail. Besides, we present how a scheduler can determine the amount of offered energy for energy transfers, and outline how a scheduler can assure that a thread receives at most the amount of offered energy assigned to it.

In order to discuss the differences between these two power limits, we distinguish the following three courses of power consumptions in the subsequent sections of this chapter:

1. A thread has continuously a power consumption below the power limit.
2. A thread has continuously a power consumption beyond the power limit.
3. A thread has partially a power consumption above the power limit.

A.1 Strict Power Limit

Case 1 A scheduler should preserve the strict power limit over the period of a thread's quantum. It inherently strictly preserves the limit, because a thread has a power consumption below the power limit.

Case 2 The power consumption would cause a constant violation of the power limit. Thus, a scheduler must throttle a thread if the thread has a power consumption beyond the power limit, in order to meet the pre-defined power limit. This throttling lasts until the thread's power consumption is below the allowed limit. During a thread's quantum, a thread's throttling and its execution can alternate several times. Hence, the average power consumption of a thread should be – after its quantum – equal to the power limit. The partially lower power consumption of the thread does not result from its execution, it is only caused by the throttling mechanism.

Case 3 In distinction to a lower power consumption caused by the throttling mechanism, a thread itself can cause its lower power consumption. It has only partially a power consumption above the allowed power limit. Depending on the course of a thread's power consumption, it is possible that the accounted power consumption never violates the power limit although it does happen, because a scheduler cannot account the power consumption constantly. In this case, the average power consumption during a thread's quantum is at most equal to the power limit, otherwise a scheduler must throttle a thread during the thread's execution. This assures that a scheduler can strictly preserve the power limit.

A.2 Non-Strict Power Limit

The major difference between the strict power limit and the non-strict power limit is the permitted energy transfer in the latter case. Therefore, we loose the requirement to preserve the power limit over a thread's quantum, and we allow to preserve the power limit over a hyper-period of several threads' quanta. Due to the maximum capacity of a run-queue's energy budget (cf. Subsection 3.4.3), the hyper-period does not need to be limited, additionally. To point out which thread can offer energy for energy transfers and how much energy a thread may receive of the offered energy, we consider the three courses of power consumption previously outlined.

Case 1 A thread has continuously a power consumption below the power limit. Consequently, a thread has not consumed as much energy as allowed by the power limit. This non-consumed energy can a scheduler now partition, offer and transfer among all threads or only among a subset of threads of the run-queue. It is defined as

$$E_{\text{offered}} = (P_{\text{limit}} - P_{\text{average}}) \cdot t_{\text{quantum}} \tag{A.1}$$

while P_{limit} is the pre-defined power consumption limit and P_{average} a thread's accounted average power consumption during its quantum t_{quantum}.

Case 2 This type of thread will have continuously a power consumption beyond the power limit if a scheduler does not throttle it. The share of the fraction or the complete fraction a thread receives of this offered energy E_{frac} may a thread only use for exceeding the power limit during its quantum. After each timer tick a scheduler accounts a thread's energy consumption $E_{\text{last tick}}$. The following equation will be true, if a scheduler does not throttle a thread of this type. It shows that the energy consumption caused within the period $t_{\text{last tick}}$ between the last and the current timer tick is greater than the allowed consumption due to the power limit:

$$E_{\text{last tick}} > P_{\text{limit}} \cdot t_{\text{last tick}} = E_{\text{limit tick}} \tag{A.2}$$

A scheduler does not need to throttle a thread as long as the thread has not consumed the complete fraction of the offered energy E_{frac} assigned to it. Therefore, a scheduler increases on each timer tick the amount of energy $E_{\text{receive}_{i-1}}$ a thread has received during the previous $i-1$ timer ticks by the received energy $E_{\text{last tick}} - E_{\text{limit tick}}$ the thread has received during its i^{th} timer tick. A thread must not receive more energy than offered to it (E_{frac}), therefore E_{receive_i} must be limited by E_{frac}, as outlined in the next equation:

$$E_{\text{received}_i} = min\{E_{\text{received}_{i-1}} + (E_{\text{last tick}} - E_{\text{limit tick}}), E_{\text{frac}}\} \tag{A.3}$$

After a thread has received its assigned fraction of offered energy and may therefore exceed the power limit no longer, its power consumption must meet the power limit. Its current power consumption is above the power limit due to the received energy which has permitted the thread to exceed the pre-defined power limit. Depending on the mechanism to account a thread's power consumption, the mechanism considers a thread's power consumption caused during the energy transfer for a thread's current power consumption. In this case, a scheduler would throttle the thread until the thread's power consumption would be beneath the power limit, but we do not desire this. Only the power consumption should be considered to meet the power limit caused after the energy transfer. Therefore, a scheduler must assure that the previous power consumption has no longer any impact on a thread's accounted power consumption.

If a thread has not completely consumed the amount of energy E_{frac} before it loses processor control, a scheduler can offer the non-consumed energy of a thread's fraction $E_{\mathrm{frac}} - E_{\mathrm{received}}$ to other threads. Furthermore, if a thread's power consumption is beyond the permitted power limit due to performed energy transfers, the thread's received energy must not cause the thread's throttling later on.

Case 3 As outlined in regard to the strict power limit, the average power consumption of a thread can be below the power limit, although a thread has partially violated the power limit. It would be a drawback for the further execution of such a thread if we assume a thread's current power consumption to be equal to the permitted power limit after its quantum. Consequently, we may only assume that a thread's power consumption is equal to the power limit if it is a benefit for a thread.

A thread of this type partially exceeds the power limit, hence equation (A.2) can be true from time to time. Only if this equation is true, the estimated power consumption exceeds the power limit and $E_{\mathrm{received}} < E_{\mathrm{frac}}$, a thread may receive energy. If this equation is true but a thread's accounted power consumption does not exceed the power limit, an energy transfer will be unnecessary.

Another difference between threads consuming steadily or only partially more power than allowed is that the average power consumption of the latter allows an energy transfer. This will be obvious for the case if a thread's average power consumption is below the power limit, in spite of performed energy transfers. Nevertheless, it is even mandatory if a thread's average power consumption is above the power limit, but it had a partial power consumption beneath the power limit as well. Otherwise, the non-consumed energy caused by a thread's power consumption below the limit would not be offered to other threads for exhausting the processor's power limit. Instead, the performed energy transfers decrease the amount of offered energy. Next, we outline how we can assure that a thread's received energy E_{receive} does not reduce the amount of energy it can offer to other threads.

In order to account that a thread can receive energy as well as offer energy, because it has partially a power consumption below the limit, we must consider a thread's received energy E_{receive} if we offer a thread's non-consumed energy. This received energy does not only have to be considered for the case of an average power consumption beyond the limit but also beneath, because the received energy has raised the average power consumption of both.

After a thread has executed for its quantum t_{quantum}, a scheduler can transfer the following energy E_{offered} to other threads:

$$E_{\mathrm{offered}} = (P_{\mathrm{limit}} - P_{\mathrm{average}}) \cdot t_{\mathrm{quantum}} + E_{\mathrm{received}} \tag{A.4}$$

In contrast to the preceding definition of E_{offered} in (A.1), we have considered the received energy. In the case of threads continuously consuming less power than allowed during their execution, their received energy E_{received} is zero. Therefore, the more general equation (A.4) applies to them.

To point out why the latter equation is necessary, we have changed thread's Th_3 course of its power consumption in comparison to the previous examples. The thread still has an average power consumption of $\frac{2L}{3}$, but at first it receives a share of thread Th_2's offered energy. In Figure A.1 we have applied equation (A.1) and in Figure A.2 we have deployed the last equation. We outline in the first example that thread Th_3 receives energy, but does not offer this received energy again. This causes that thread Th_4 is throttled more often in comparison to the latter example, which considers the received energy.

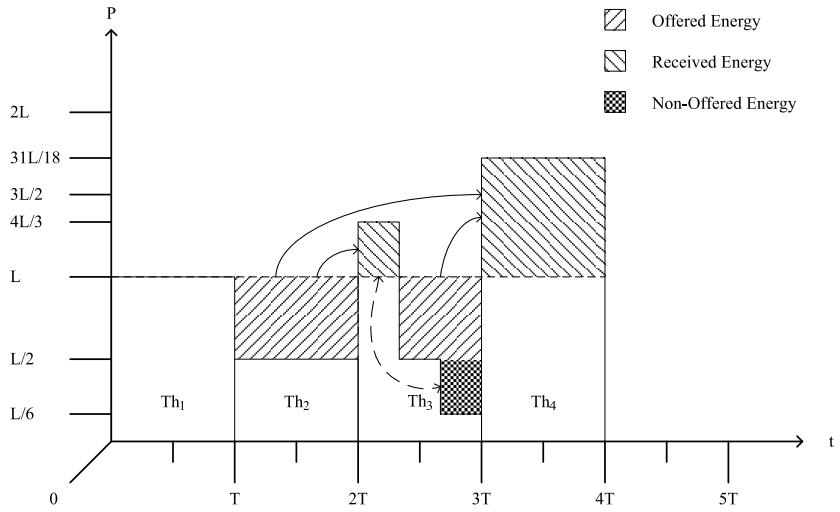

Figure A.1: Non-Strict Power Limit Without Considering Received Energy

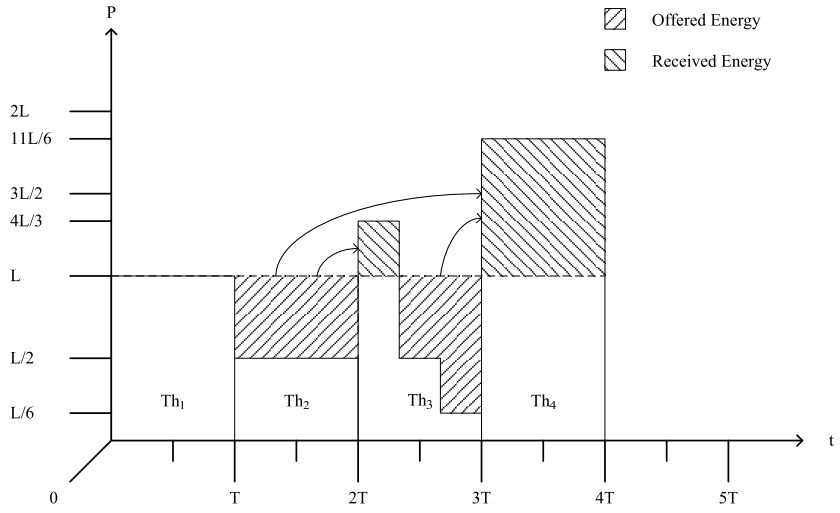

Figure A.2: Non-Strict Power Limit With Considering Received Energy

Appendix B

Run-Queue Energy Budget

We have proposed in Subsection 3.4.2 a design to partition the offered energy fairly among threads. To partition the offered energy E_{offered} only among threads eligible to be scheduled and willing to receive the energy, we have restricted the number of receiving threads to $\text{threads}_{\text{receive}}$ as outlined in Subsubsection 3.4.2.1. If we do not adjust the number of receiving threads, a scheduler will partition the offered energy among all n threads of a run-queue. Therefore, even if the offered energy is sufficient to fulfill each thread's request to receive the amount of offered energy to achieve its best performance, a scheduler cannot fulfill the requests after the first run, because it partitions the energy among all threads of a run-queue. Consequently, the receiving threads have only consumed a fraction of the offered energy after a run, the remaining offered energy of a run offers a scheduler during the next run. We will prove at next that at the latest after n^2 runs – n is the number of threads of a run-queue – a scheduler can fulfill each thread's request if the offered energy is sufficient to fulfill their requests.

Theorem If a scheduler fairly partitions the offered energy, as outlined in Subsection 3.4.2, and the offered energy is sufficient to fulfill each thread's request to receive the amount of offered energy to achieve its best performance, a scheduler can fulfill each thread's request at the latest after n^2 runs.

Proof We assume without loss of generality (w.l.o.g.) that the threads Th_1 to Th_k offer energy and the threads Th_{k+1} to Th_n receive the offered energy. The offered energy is sufficient to fulfill the receiving threads requests. To assure that a scheduler has not to offer more energy as later on received by the requesting threads, we assume that the offered energy E_1 up to E_k of the first k threads is equal to the requested energy E_{k+1} up to E_n of the last $n - k$ threads:

$$E_{\text{offered}} = \underbrace{E_1 + \cdots + E_k}_{k \text{ threads offering energy}} = \underbrace{E_{k+1} + \cdots + E_n}_{n-k \text{ threads receiving energy}} \tag{B.1}$$

After the first run, the threads Th_{k+1} to Th_n have consumed E_{consumed_1} energy of the offered energy. A scheduler partitions the energy among all n threads, but not only

among the $n-k$ threads requesting the energy, therefore it is not possible to fulfill each thread's request.

$$\begin{aligned} E_{\text{offered}} &\geq \min_{k+1\leq i\leq n}\{E_i, \frac{E_{\text{offered}}}{n}\}(n-k) \\ &= E_{\text{consumed}_1} \end{aligned} \tag{B.2}$$

In order to prove the correctness of our theorem, we have to distinguish between the two borderline cases:

1. $(n-k-1)$ requests can be fulfilled, but one request cannot be fulfilled.
2. $(n-k)$ requests cannot be fulfilled.

These two borderline cases are important, because they will reveal the worst case number of runs required to fulfill each thread's request.

Case 1 W.l.o.g. we assume that a scheduler can fulfill the requests of the threads Th_{k+1} up to Th_{n-1}. Only thread's Th_n request cannot be fulfilled by a scheduler. Additionally, we assume that only thread Th_n requires energy. This can happen if the threads Th_{k+1} up to Th_{n-1} have power consumptions equal to the power limit and neither offer nor receive energy. Therefore, this permits to evaluate the worst case number of runs until a scheduler can fulfill each thread's request:

$$E_{k+1} = \cdots = E_{n-1} = 0, E_n = E_{\text{offered}} \tag{B.3}$$

Run 1 Thread Th_n has consumed after the first run only E_{consumed_1} of the offered energy.

$$\overset{(B.3)}{\Rightarrow} E_{\text{consumed}_1} = \frac{E_{\text{offered}}}{n} \tag{B.4}$$

Run 2 The energy $E_{\text{offered}} - E_{\text{consumed}_1}$ which the thread has not received after the first run, offers a scheduler additionally to the offered energy designated for the second run E_{offered} to the thread.

$$\begin{aligned} 2E_{\text{offered}} - E_{\text{consumed}_1} &\geq \min_{k+1\geq i\geq n}\{E_i, \frac{2E_{\text{offered}} - E_{\text{consumed}_1}}{n}\} \\ &\cdot(n-k) = E_{\text{consumed}_2} \end{aligned} \tag{B.5}$$

Although thread Th_n is the only thread receiving energy, it has not received its requested energy E_n. Instead, it has only consumed the following energy in the second run:

$$\begin{aligned} \overset{(B.3)}{\Rightarrow} E_{\text{consumed}_2} &= \frac{2E_{\text{offered}} - E_{\text{consumed}_1}}{n} \\ &= \frac{2E_{\text{offered}} - \frac{E_{\text{offered}}}{n}}{n} \\ &= \frac{2E_{\text{offered}}}{n} - \frac{E_{\text{offered}}}{n^2} \\ &= \sum_{i=1}^{2} \left(\begin{array}{c} 2 \\ i \end{array} \right) \frac{E_{\text{offered}}}{n^i}(-1)^{(i+1)} \\ &= E_{\text{offered}} - E_{\text{offered}} \left(\frac{-1+n}{n} \right)^2 \end{aligned} \tag{B.6}$$

Run n^2 After $n(n-1)$ runs, thread Th_n has already received $\sum_{j=1}^{n(n-1)} E_{\mathrm{consume}_j}$ energy of the $n(n-1)E_{\mathrm{offered}}$ amount of offered energy. In run n^2, thread Th_n receives the energy $E_{\mathrm{consume}_{n^2}}$.

$$\begin{aligned} \underbrace{n^2 E_{\mathrm{offered}} - E_{\mathrm{consumed}_1} - \cdots - E_{\mathrm{consume}_{n(n-1)}}}_{\alpha} &\geq \min_{k+1 \leq i \leq n} \{E_i, \frac{\alpha}{n}\}(n-k) \\ &= E_{\mathrm{consume}_{n^2}} \end{aligned} \tag{B.7}$$

If we assume that the thread Th_n has still not consumed its requested energy E_n, $E_{\mathrm{consume}_{n^2}}$ is:

$$\begin{aligned} \overset{(B.3)}{\Rightarrow} E_{\mathrm{consumed}_{n^2}} &= \sum_{i=1}^{n^2} \left(\begin{array}{c} n^2 \\ i \end{array} \right) \frac{E_{\mathrm{offered}}}{n^i} (-1)^{(i+1)} \\ &= E_{\mathrm{offered}} - E_{\mathrm{offered}} \left(\frac{-1+n}{n} \right)^{n^2} \end{aligned} \tag{B.8}$$

After n^2 runs, a scheduler can fulfill the thread's request of receiving energy E_n.

$$\lim_{n \to \infty} \left(E_{\mathrm{offered}} - E_{\mathrm{offered}} \left(\frac{-1+n}{n} \right)^{n^2} \right) = E_{\mathrm{offered}} = E_n \tag{B.9}$$

Consequently, also a scheduler can fulfill the thread's requests of the subsequent runs.

Case 2 A scheduler cannot fulfill the requests of the threads Th_{k+1} up to Th_n. W.l.o.g. we assume that each of these threads Th_i requires the same amount of energy. This permits to evaluate the worst case number of runs until a scheduler can fulfill each thread's request:

$$E_{k+1} = \cdots = E_n = \frac{E_{\mathrm{offered}}}{n-k} \tag{B.10}$$

Run 1 After the first run, the receiving threads have received together $E_{consumed_1}$ energy of the offered energy.

$$\overset{(B.10)}{\Rightarrow} E_{\mathrm{consumed}_1} = (n-k)\frac{E_{\mathrm{offered}}}{n} \tag{B.11}$$

Run 2 The energy $E_{\mathrm{offered}} - E_{\mathrm{consumed}_1}$ which the threads have not received after the first run, offers a scheduler in addition to the offered energy designated for the second run E_{offered} to the threads.

$$\begin{aligned} 2E_{\mathrm{offered}} - E_{\mathrm{consumed}_1} &\geq \min_{k+1 \geq i \geq n} \{E_i, \frac{2E_{\mathrm{offered}} - E_{\mathrm{consumed}_1}}{n} \} \\ &\quad \cdot (n-k) = E_{\mathrm{consumed}_2} \end{aligned} \tag{B.12}$$

The receiving threads have not received their requested energy $\frac{E_{\text{offered}}}{n-k}$ in this run, therefore they have only consumed the following energy:

$$\begin{aligned}
\overset{(B.10)}{\Rightarrow} E_{\text{consume}_2} &= \frac{2E_{\text{offered}} - E_{\text{consume}_1}}{n}(n-k) \\
&= \frac{2E_{\text{offered}}}{n}(n-k) - \frac{E_{\text{offered}}}{n^2}(n-k)^2 \\
&= \sum_{i=1}^{2} \binom{2}{i} \frac{E_{\text{offered}}(n-k)^i}{n^i}(-1)^{(i+1)} \\
&= E_{\text{offered}} - E_{\text{offered}} \left(\frac{k}{n}\right)^2 \qquad \text{(B.13)}
\end{aligned}$$

Run n After $n-1$ runs, the threads Th_k to Th_n have already received the energy $\sum_{j=1}^{n-1} E_{\text{consume}_j}$ of the amount of offered energy $(n-1)E_{\text{offered}}$. In run n, they receive the energy E_{consume_n}.

$$\begin{aligned}
\underbrace{nE_{\text{offered}} - E_{\text{consumed}_1} - \cdots - E_{\text{consume}_{n-1}}}_{\alpha} &\geq \min_{k+1 \leq i \leq n} \{E_i, \frac{\alpha}{n}\}(n-k) \\
&= E_{\text{consume}_n} \qquad \text{(B.14)}
\end{aligned}$$

If we assume that the threads Th_{k+1} to Th_n have still not consumed their requested energy $\frac{E_{\text{offered}}}{n-k}$, E_{consume_n} is:

$$\begin{aligned}
\overset{(B.10)}{\Rightarrow} E_{\text{consume}_n} &= \sum_{i=1}^{n} \binom{n}{i} \frac{E_{\text{offered}}(n-k)^i}{n^i}(-1)^{(i+1)} \\
&= E_{\text{offered}} - E_{\text{offered}} \left(\frac{k}{n}\right)^n \qquad \text{(B.15)}
\end{aligned}$$

After n runs, a scheduler can fulfill each thread's request to receive the energy $\frac{E_{\text{offered}}}{n-k}$.

$$\lim_{n\to\infty} \left(E_{\text{offered}} - E_{\text{offered}} \left(\frac{k}{n}\right)^n \right) = E_{\text{offered}} = \sum_{i=k+1}^{n} E_i \qquad \text{(B.16)}$$

Thus, a scheduler can fulfill the subsequent requests of the threads Th_{k+1} to Th_n after n runs.

In summary, our proposed energy transfer assures that a scheduler can fulfill at the latest after n^2 runs each thread's request to receive the thread's required energy if the offered energy is sufficient to fulfill each thread's request.

Bibliography

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual. Volume 2. System Programming*, 3.14 edition, September 29 2007.

[2] Amazon elastic compute cloud. `http://www.amazon.com/gp/browse.html?node=201590011`, July 04 2008.

[3] K. Banerjee, Sheng-Chih Lin, A. Keshavarzi, S. Narendra, and V. De. A self-consistent junction temperature estimation methodology for nanometer scale ics with implications for performance and thermal management. *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, pages 36.7.1–36.7.4, Dec. 2003.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

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

[6] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, New Orleans, LA, September 27 2003.

[7] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3. ed. edition, 2006.

[8] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182, 2001.

[9] Completely fair scheduler. `http://kerneltrap.org/node/8059`, April 18 2007.

[10] P. Dadvar and K. Skadron. Potential thermal security risks. *Semiconductor Thermal Measurement and Management Symposium, 2005 IEEE Twenty First Annual IEEE*, pages 229–234, March 2005.

[11] D.J. Deleganes. A high performance, low power pentium processor. *Circuits and Systems, 1995., Proceedings., Proceedings of the 38th Midwest Symposium on*, 2:1127–1130 vol.2, Aug 1995.

[12] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Comput. Archit. News*, 34(2):78–88, 2006.

[13] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. *SIGOPS Oper. Syst. Rev.*, 36(SI):105–116, 2002.

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

[15] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[16] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. pages 491–505, 2001.

[17] Wei Huang, Mircea R. Stan, Kevin Skadron, Karthik Sankaranarayanan, Shougata Ghosh, and Sivakumar Velusam. Compact thermal modeling for temperature-aware design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 878–883, New York, NY, USA, 2004. ACM.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3B. System Programming Guide, Part II*, February 2008.

[19] Intel Corporation. *Intel® Core™ 2 Duo Processor and Intel® Core™ Extreme Processor on 45-nm Process for Platform Based on Mobile Intel® 965 Express Chipset Family*, January 2008. Data Sheet.

[20] Simon Kellner. Event-driven temperatur control in operating systems. Study thesis, Operating System Group, University of Erlangen, Germany, April 30 2003.

[21] A. Kumar, Li Shang, Li-Shiuan Peh, and N.K. Jha. System-level dynamic thermal management for high-performance microprocessors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(1):96–108, Jan. 2008.

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

[23] Huan Liu and Dan Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *CCGRID*, pages 295–305. IEEE Computer Society, 2008.

[24] Thomas L. Martin. *Balancing batteries, power, and performance: system issues in cpu speed-setting for mobile computing*. PhD thesis, Pittsburgh, PA, USA, 1999. Adviser-Daniel P. Siewiorek.

[25] Andreas Merkel. Balancing power consumption in multiprocessor systems. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, September 30 2005.

[26] Ingo Molnar. Completely fair scheduler patch. `http://people.redhat.com/mingo/cfs-scheduler/sched-cfs-v2.6.22.15-v24.1.patch`, January 14 2008.

[27] Netperf benchmark. `http://www.netperf.org/`, July 14 2008.

[28] C.D. Patel, R. Sharma, C.E. Bash, and A. Beitelmal. Thermal considerations in cooling large scale high compute density data centers. *Thermal and Thermomechanical Phenomena in Electronic Systems, 2002. ITHERM 2002. The Eighth Intersociety Conference on*, pages 767–776, 2002.

[29] C. Poellabauer, L. Singleton, and K. Schwan. Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications. *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 234–243, March 2005.

[30] Anand Raghunathan, Niraj K. Jha, and Sujit Dey. *High-Level Power Analysis and Optimization*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[31] Ravishankar Rao and Sarma Vrudhula. Performance optimal processor throttling under thermal constraints. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 257–266, New York, NY, USA, 2007. ACM.

[32] E. Rohou and M. Smith. Dynamically managing processor temperature and power. In *2nd Workshop on FeedbackDirected Optimization*, Nov 1999.

[33] A. Shah, V. Carey, C. Bash, and C. Patel. Impact of chip power dissipation on thermodynamic performance. *Semiconductor Thermal Measurement and Management Symposium, 2005 IEEE Twenty First Annual IEEE*, pages 99–108, March 2005.

[34] K. Shin and T. Kim. Leakage power minimisation in arithmetic circuits. *Electronics Letters*, 40(7):415–417, April 2004.

[35] Spec cpu2006. `http://www.spec.org/cpu2006/`, July 14 2008.

[36] Inc. Sun Microsystems. On-demand computing using network.com. White paper, Santa Clara, CA, USA, August 2007.

[37] Andrew S. Tanenbaum. *Modern operating systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2001.

[38] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. *Design Automation Conference, 1998. Proceedings*, pages 732–737, Jun 1998.

[39] Lisa A. Torrey, Joyce Coleman, and Barton P. Miller. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Softw. Pract. Exper.*, 37(4):347–364, 2007.

[40] Martin Waitz. Accounting and control of power consumption in energy-aware operating systems. Diploma thesis, Operating System Group, University of Erlangen, Germany, January 31 2003.

[41] C. A. Waldspurger and E. Weihl. W. Stride scheduling: Deterministic proportional- share resource management. Technical report, Cambridge, MA, USA, 1995.

[42] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1, Berkeley, CA, USA, 1994. USENIX Association.

[43] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[44] Andreas Weissel and Frank Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246, New York, NY, USA, 2002. ACM.

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

[46] Jun Wu and Tei-Wei Kuo. Real-time scheduling of cpu-bound and i/o-bound processes. *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 303–310, 1999.

[47] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.