Diplomarbeit

Towards Consolidate-to-Idle

Johannes Steinmetz

14. Mai 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:

Prof. Dr. rer. nat. Hermann Härtig

Betreuende Mitarbeiter:

Dipl.-Inf. Marcus Hähnel
Dr.-Ing. Marcus Völp

Aufgabenstellung

Race-to-Idle, that is the quick execution of tasks to later on be able to power down processors, is a well understood principle for energy efficient scheduling of real-time tasks. The goal of this diploma thesis is to prepare a translation of this scheduling scheme into the spatial dimension. Rather than executing tasks on all processors, tasks should, whenever possible, be consolidated on a few active cores, allowing the remaining ones to be deactivated in the mean time. For this, the costs (both energy and latencies) for powering up and down processors must be determined, migration costs and the follow on costs from such a migration must be analyzed and application progress must be tracked. All this data can then guide scheduling and load balancing decisions to determine when additional processors are required or when it is possible to reconsolidate the current workload. The expected outcome of this thesis is a profound analysis of the information required to drive these decisions, which ideally, but not necessarily, is complemented by a first exploration of Consolidate-to-Idle load balancing.

Erklärung

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

Dresden, den 14. Mai 2013

Johannes Steinmetz

Danksagung

Ich bedanke mich recht herzlich bei allen Mitarbeitern der Dresdner Betriebssystemgruppe von Professor Härtig. Besonders meine Betreuer haben immer eine offen Tür gehabt aus der man mit Rat und interessanten neuen Fragen heraustreten konnte. Ich habe hier in sehr viel lernen können, nicht nun über Betriebssysteme.
Weiterhin bedanke ich mich bei allen Entwicklern, die mit freier Software dafür sorgen, dass neugierige IST-Studenten und Informatiker immer genug zu studieren haben und sie mit Software-Werkzeugen vom Feinsten ausrüsten.
Ich danke auch ganz herzlich meinen Freunden, die mich die letzen Wochen ertragen haben, mich motiviert haben oder sogar einige verlorene Kommata aufgelesen und in meinen Computer zurückgetragen haben.

Mein größter Dank gebührt meiner Familie, die in jeder Hinsicht immer für mich da und ganz großartig ist. DANKE!

Contents

1 Introduction

Real-time systems form an important part of todays landscape of digital programmable computers since time is an important part of real-world problems: From telecommunication systems with various signal processing steps involved, over traffic regulation systems, medical health monitoring or power plant regulationj Diverse applications that demand problem solutions in specific timely bound manner are ubiquitous. Solving these problems with computers needs operating systems and application software that is aware of the timing component of the computations involved. Time is a first class resource for such systems—just like memory or algorithmic processing units—and in complex real-time systems it is usually the role of an operating system to partition and share all computationally relevant resources between the sub-tasks of the system.

The component of such a system that does the time-slicing and allocation of processing units to parts of the task is usually called the CPU-scheduler or just scheduler and is an important part of a real-time operating system for time constrain adherence. This thesis' purpose is the beginning of the exploration of a new scheduling paradigm in the domain of real-time computing.

Unlike other real-time scheduling approaches, it focuses on two extra qualities beside the real-time typical timing guarantees: First, it addresses the need for speedup that nowadays is mostly made in spatial dimension: Computer technology slowed down upscaling the clock frequency of computerchips, but improvements in decreasing the chip area consumption of processing units let the manufacturers compensate this by implementing more units on a chip instead of faster ones. This has led to multi core processors even in embedded signal processing [9], and mobile applications like contemporary smart-phones [16].

The second dimension of efficiency has always been a number one concern in embedded and mobile applications and is constantly growing in importance in all fields of computing: Energy consumption. Longer usage intervals without dependency on stationary power supply opens up mobile computing to new fields of application and improves usefulness in existing ones. Also, less energy consumption leads to less heat that has to be dissipated off the computer chip and therefore a potential improvement of package or heat-sink size, the maximum possible clock frequency or the operating lifetime of computer components.

The *Consolidate-to-Idle* idea discussed in this thesis raises hope that in a multi processing unit environment for some real-time applications, such energy economi-

sation could be achieved by only software and therefore almost for free. To study the behaviour of *Consolidate-to-Idle* on real hardware and to have a prototype for future implementations on different operating systems, I built an implementation of *Consolidate-to-Idle* as part of a system running with the well known Linux kernel.

But before I will describe the details for a *Consolidate-to-Idle* scheduler design in Section 3, I will take a look in Section at the basics and souroundings of real-time scheduling to lay the foundation for an energy aware real-time scheduling paradigm 2. Afterwards in Section 4 I will get into implementational details and show how I mapped the design into a working CPU-scheduler running in the Linux kernel. The Section 5 following will evaluate the practical implementations with measurements and their discussion. In the concluding Section 6 I will give a résumé and take a look-out on possible follow-up work on *Consolidate-to-Idle*.

2 Foundations and Related Work

The following section is meant to build up or refine the foundation for my research. After a definition of the most relevant technical terms, I will explain basic concepts that are relevant in the context of a *Consolidate-to-Idle* scheduler design. Those are the task model, that I will apply, the *Race-to-Idle* concept as the alternative to *Consolidate-to-Idle*, *Earliest Deadline First (EDF)* scheduling as the choosen basic scheduling algorithm and the Linux kernel scheduling architecture, which will house the implementation of my *Consolidate-to-Idle* scheduler described later or in Section 4. The section closes with a brief view on related works, concerning alternate approaches in the field of energy efficient scheduling.

2.1 Terminology

For reference and clarification, I will give a definition for the most important technical terms of scheduling theory in the following subsection, as far as they are relevant in the exploration of the *Consolidate-to-Idle* scheme or in the code of the research implementation that is part of this thesis. Different definitions in subtile aspects and most notably symbol names of those terms are in use in the scheduling literature. I try to stick to the notation that is used by Liu in her "Real Time Systems" book [10] as it covers the aspects I will need in the design as well.

Processing Unit As long as a computer had just one unit, that actually decodes and executes the main computer program, it was pretty clear that this part of the machine could be called the *central* processing unit (CPU). Whether the actual hardware chip or the more abstract logical concept from the programmers point of view was taken: The term CPU could have been used for both, given a proper context.
Nowadays computer hardware is different: A single chip contains multiple units, that may execute code independently. Those *processor cores* may also have the hardware capability to provide more than one path of decoding and executing instructions on their arithmetic and logical units, so that those processing units operate on a higher level of utilization, while other resources remain shared between those units. Intel's *hyperthreading* technology as presented in [12] gives an example for this. Thus—from an abstract point of view—each single processing

core may provide multiple execution paths.

A computer may in addition contain more than just one chip for code execution and all units may or may not be equally equipped with execution capabilities.
For this thesis a more logical view is applied to model units that are capable of executing code. Processor, processing unit or processor core are used interchangeably here and overall abbreviated with the well coined abbreviation CPU. For the mathematical description, with a number n of CPUs, I will symbolize those execution units with $P_i \quad (i \in \mathbb{N}, 1 \leq i \leq n)$.

Task Task is a term from the domain of scheduling theory. A task is a stream of jobs logically connected. It describes a more abstract model of some work that has to be dealt with by the computer program, in contrast to the technical terms*process* or *thread* used in operating systems.

Job A job is some specific part of the workload belonging to a task. Each task might consist of several jobs. If all requirements in resources and preconditions of a job are fulfilled the job becomes eligible to run on a processing unit. The job is called to be *ready* then. The point in time when a job J_i gets ready for the first time is the *release time* r_i or *spawn time* of the job. When the operating system selects a *ready* job to run on a CPUs it becomes *running* and executes on the CPUs. If additional resources or conditions are required during the computation of the job, the *ready*-state may be interrupted. This state change is called blocking. A running job may also block voluntarily during runtime by requiring a time related condition. When all of a job's computation has been done, the job has been *completed*. The amount of processing time resources that were used on a job's completion is its *execution time* e_i. A task may have periodic jobs, which are jobs that are released by their task in a specific constant amount of time after their previous release.

Worst-Case Execution Time The time resources needed to complete a job J_i are usually not a priori known and may vary in an interval from the minimum execution time e_i^- up to the maximum of all possible execution times the *worst-case execution time (WCET)* e_i^+. The expected average execution time might be well below the WCET e_i^+, especially if the WCET has to be estimated by empirical heuristics and is impinged with safety margins to buffer imperfect system

behaviour.

Real-Time A task is considered to be a real-time task, if correctness is not only expected in the logical result of the computation, but also in timing behaviour. Usually timing constrains are expressed by assigning the jobs J_i of the real-time task some *deadline* d_i, a specific instant in time by which a job has to be completed.

Hard- and Soft- Real-Time Requirements Real-time requirements can be divided in two main categories in respect of their deadline exhaustion behavior or the so called *laxity type* of the jobs. In hard real-time systems a task may never exceed the time limit given by its deadline. Otherwise complete system failure would be the result.
In soft real-time systems the requirements are not that strict; a late result may have to be dismissed or it leads to inferior system quality, but the system as a whole is still to be considered operational.
A good example for soft real-time requirements is a video decoding system: Each video frame has to be decoded in a specific amount of time to achieve the desired image presentation frequency. Missing a few deadlines would result in lower perceived video quality, which could be tolerable, or even barely noticeable, if it happens seldom enough. In contrast to this soft real-time requirement a late result in a control loop done by a signal processor that is part of the avionic of a military airplane might cause the plane to crash. As this is absolutely not tolerable it has to be seen as a complete system failure. The control loop system is thus a hard real-time system.
Soft real-time systems may also specify statistical constrains on the number of missed deadlines or their frequency.

Slack time If at the current time t_c the execution of a job J_i could be delayed by the scheduler by some amount of time without compromising the deadline of any job , this amount is called the *slack time*, or in short just "slack", of the job $\sigma_i(t_c)$. The amount of time that all the jobs of a task set on the whole could be feasibly delayed—withouth violation of *any* deadline on the system—is called the system slack $\sigma(t_c)$.

2.2 Periodic Task Model

For scheduling purposes one has to find a task model that characterizes the tasks to be scheduled in an application independent way.
A well-known and deterministic workload model is described by Liu in [10, Chapter 3]: The model of periodic tasks, which is focused on for the rest of this thesis. A task set in this model consists of periodic tasks, which means that tasks spawn a job repeatedly with a regular time interval or a semi-regular time interval where some time variation, called jitter, is allowed to happen. Each job can only run on one processing unit at a time, so that beneficial parallel execution could be done only with multiple jobs. Each job belongs to a task, that is statically existent after task creation in the system forever. For simplification, the jobs are assumed to be independent from each other and when they become ready they are assumed to need only enough CPU time to complete. This actually means that the jobs are laid out in an non-blocking manner, so that they can complete in their bounded execution time in one piece of time, as long as they are assigned to aCPU.

For the symbolic description in this thesis it suffices to describe the periodic task as a triplet consisting of the tree values that are not implicitly given from the later refinement of the task model: WCET e_i^+, the period length, and the offset of the first period to the beginning of a hyperperiod when more tasks are concerned in a task set.

2.3 Race to Idle

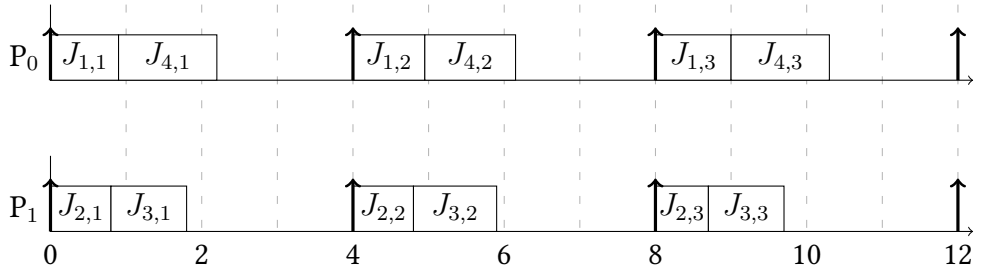


Figure 1: A sample task set of four periodic tasks with a period of 4 scheduled in a Race-to-idle manner :$\{T_1(1,4,0), T_2(1,4,0)T_3(1.4,4,0), T_4(1.3,4,0)\}$.

Race-to-idle is a well known standard scheduling paradigm used in multiprocessor real-time schedulers. The basic idea is to maximize parallel execution by using

all applicable resources as early as reasonably possible. The more resources are provided, the earlier all tasks are done and the resources can be used for non real-time tasks oder be switched off for energy consumption optimisation. Figure 1 depicts an example schedule for a task set on a system with 2 CPUs.

2.4 Earliest Deadline First

Earliest Deadline First (EDF) is a particular scheduling strategy. All runnable jobs of a task set get assigned a dynamically changing priority. The nearer the deadline of a particular job is in time, the higher is the priority the job is assigned to. In other words: The job with the earliest deadline gets scheduled on the CPU first.

Optimal in the context of scheduling strategies means that an optimal strategy will always find a feasible schedule if—and only if—such a schedule exists for the given task set. It can be shown, that earliest-deadline-first (EDF) with preemptible jobs is optimal for an uniprocessor system. In [10, Chapter 4.6] Liu shows such a proof for optimality on uniprocessor EDF, that is based on the systematically transformation of any feasible schedule into a schedule that is found by an EDF scheduler.

That an EDF scheduler is not optimal in a multiprocessor system can be shown simpy with an example from [10, Chapter 4.7]: Given three jobs J_1, J_2 and J_3 with WCETs of 1, 1 and 5 and deadlines of 1, 2 and 5 that are all released at $t_0 = 0$ a feasible schedule exists as depicted in Figure 2b. Since J_3 has the latest deadline it features the lowest priority such that J_1 and J_2 would run parallel on P_1 and P_2 scheduled by EDF. J_3 would miss its deadline as shown in Figure 2a.

2.5 Linux Kernel Scheduling Architecture

Since my example implementation will be done in the Linux kernel (see Section 4.1), I will give an overview of the Linux kernels scheduling architecture. The Linux task and scheduling framework builds the foundation of my implementation that is further described in Section 4.

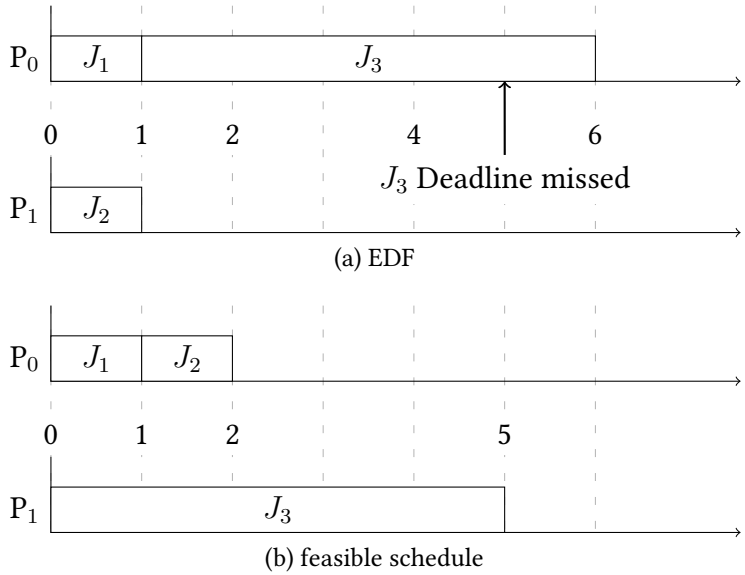


Figure 2: A sample job set that demonstrates nonoptimality of the EDF algorithm in a multiprocessor system. (a) Is the schedule according to EDF whereas (b) shows that a feasible schedule exists.

2.5.1 Modular Scheduler

Since its version 2.6.23[1] the Linux kernel features a modular scheduler: The main mechanisms for scheduling are provided by the scheduling core whereas policies for scheduling are provided by isolated scheduling classes. The main entry point for the scheduling code is the scheduling core function `schedule()` that goes through the whole hierarchy of available scheduling classes. Each class from the top of the hierarchy gets asked to return the next task, that shall be scheduled on the respective CPU on which `schedule()` got called until one class returns a runnable task.

`pick_next_task` For this purpose each class has to provide the `pick_next_task` callback. Therefore this is the main entry into the scheduling

[1] Clarification for the history interested: The article referenced by [11] is a permanent copy of a Linux kernel mailing list discussion, that lead to the inclusion of scheduler code with its then new *Completely Fair Scheduler* and a modularized core, that is a direct anchestor of the contemporary version. The given version number is not the first for which such code was discussed, but to the best of my knowledge the version when it entered mainline kernel.

class interface. But there are some more callbacks that are essential to the core interface and therefore mandatory for each existing and new scheduling class:

`enqueue_task` This gets called by the scheduling core to hand over a thread into the data structures of a scheduling class. In the normal Linux semantics a thread is afterwards *on a runqueue* (see below) and therefore ready to be executed. Usual a thread is enqueued when it is newly created, newly put on an other scheduling class, or when it gets ready after it was blocked for a while.

`dequeue_task` When a thread is blocking or finishes it work and calls for destruction, the core calls this callback to remove a thread from its runqueue.

Depending on the implemented policy a scheduling class may depend on correct implementation of some of the optional callbacks:

`put_previous_task` Gets called by the core to advise a scheduling class, that the currently running thread gets preempted an put down.

`check_preempt_curr` If a thread gets ready on the same CPU as the current thread, the core uses this callback to inform the scheduling policy class of the current thread about this event. This may lead the class code to ask for a reschedule in the return path of the callback through the core.

2.5.2 Linux Scheduling Data Structures

There are different types of data structures involved in scheduling in Linux. For an easier insight in the scheduling related code, I will list some of them in the following section.

Task structure Each thread in Linux is represented by a structure called `task_struct`. The name already mentions, that a operating system thread often represents a task practically.

Runqueue Runqueues are—besides `task_struct` structures—the main data structures of scheduling in Linux. Each CPU has its associated runqueue and all information about the threads assigned to a CPU live in this structure. There are

substructures in the runqueues for different scheduling classes which may have substructures for levels of execution priority themselves. Each thread that is running or ready is listed in exactly one runqueue.

Scheduling Entity Scheduling entities are the class specific data structures that hold the per process private data of a thread. Their exact purpose is thereby defined by each particular scheduling class and depends on the data needed to provide the policy a class implements. Scheduling entity structs are, as they are subparts of the `task_struct`, allocated for all possible scheduling classes at once during the creation of a new thread. It is therefore beneficial for scalability to keep this part small and all information of general use shared between scheduling classes in the `task_struct`.

2.5.3 Default Scheduling Classes

Currently, in contemporary versions of the Linux kernel, there are five scheduling classes, that are regularly distributed as parts of the kernel and not subject to user selection during kernel configuration. Figure 3 depicts the order in which the classes are checked for returning the next task to be scheduled. I will give a short description of them in order to give the context for any new scheduling class.

CFS Scheduler The *Completely Fair Scheduler (CFS)* is the default scheduling class of a normal task in Linux. It replaced the former *O(1)* scheduler since the modularisation of the scheduler with the kernel version 2.6.23 [11, +follow ups]. Its purpose is to maintain fair distribution of CPU time among the tasks in the system. CFS features tree flavours that can be selected as different scheduling classes from the unprivileged user mode via the `sched_setscheduler` systemcall. As those flavours are just different parametrisations of CFS they count as just one class in this case. Namely the flavours are:

- `SCHED_OTHER`, the standard round-robin time-sharing policy.
- `SCHED_BATCH`, for "batch" style execution of processes.
- `SCHED_IDLE`, for running very low priority background jobs.

CFS is not capable of real-time scheduling and thus not further relevant for this thesis. But as the most complex scheduler using the scheduling core, it provides priceless documentation for the scheduling core semantics.

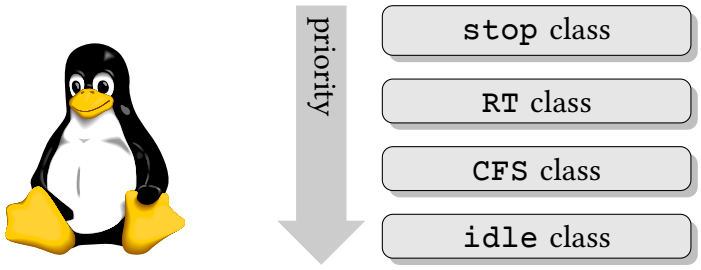


Figure 3: Hierarchie of standard scheduling policy classes in the Linux kernel from highest to lowest.

RT Scheduler The vanilla Kernel has two real-time sched classes: *Fifo*, a simple first-in-first-out strategy, and *RR*, a round-robin version. *RR* plans the task execution with timeslots of fixed length during which a task is not preempted. After timeslot exhaustion the next task of the same priority gets its timeshare until all tasks of the priority level have dequeued from the runnable task set. *Fifo* does not apply timeslot based CPU multiplex on a task set. The first task in the runqueue of highest priority gets the CPU until it dequeues. When a task becomes runnable it gets enqueued after all others of the same priority.
Those two scheduling classes leave the problems of task starvation completely to the system user. Thus, the system designer can not put uncooperative tasks on this scheduling class. Their simple timing behaviour has better predictability and makes those classes usable in some soft real-time applications.

stop class This is a special class used internally in the Linux kernel. It is the class of the highest priority. Its special purpose is to do task migration, e.g. when all tasks have to be migrated of a CPU for its deactivation. The *stop* class is not exposed to be selected for a thread from userland.

Idle The scheduling class *idle* is of lowest priority. If no other scheduling class returns a task to be run this class will always return as a last resort the special *idle* task. In this task energy optimisation is implemented by putting the concerned CPU in less power consuming states. How this is achieved depends on the exact CPU used in a system and might involve a combination of frequency and voltage downscaling or an at least partial deactivation of a whole CPU core.

2.6 Related Work

Real-time scheduling has seen many research over the last decades and also for energy efficient systems many research results have been published. A comprehensive state of the art analysis for the field of multiprocessor real time scheduling is given by Davis and Burns [4] updated 2011. In [5] Dudani et. al. combine the two aspects and use system slack for energy conservation on a uniprocessor real-time system. But unlike my approach they explore the frequency scaling abilities of processors for the conservation of power. This is also done by Chen,Yang and Kuo in[2]. In their work they focus on the slack reclamation algorithm in an multiprocessor task model.

3 Design

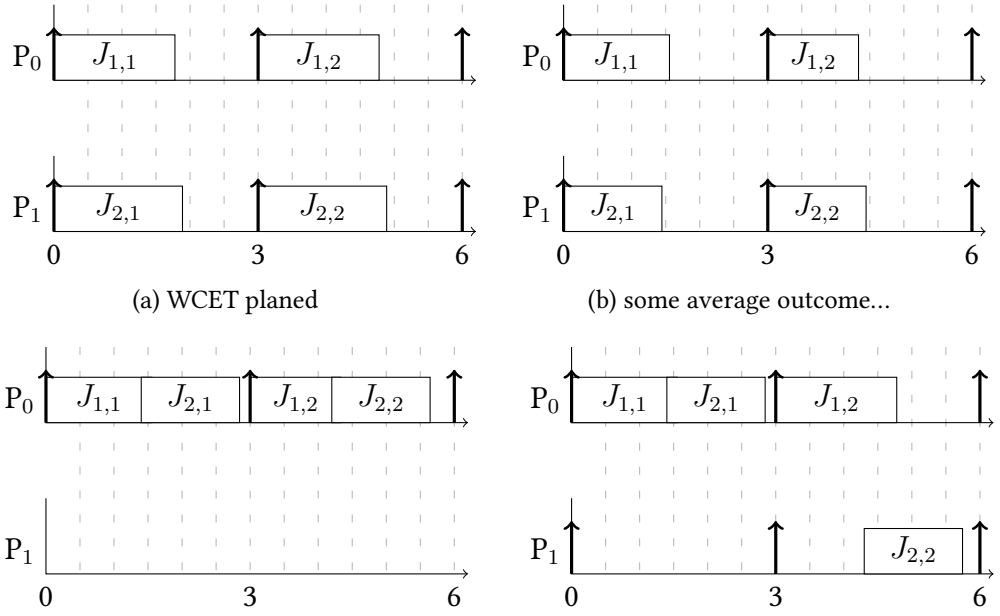
A totally static system, where an a priori known schedule is applicable a strategy of consolidation is of no special use.The static schedule could find the best way to fill one CPU up with work, that others may sleep as long as the static task set allows. But starting with slack that is reclaimed from the difference between WCET and actual completion time, even with a static schedule the idea to use slack time for consolidation on less CPU cores is interesting. The idea in this thesis is to achieve best consolidation on as few cores as possible in a dynamic, deadline driven real-time system.
This design section starts with an elaboration of the main concept of the *Consolidate-to-Idle* scheduling paradigm. Then, the concrete algorithm later used in the implementation is designed. I will close this section with an presentation of my approach, how to compensate design induced overhead for deadline safety during WCET estimation.

3.1 Consolidate-to-idle Concept

The main idea of *Consolidate-to-Idle* is to consolidate all work to as few CPUs as possible and thus avoiding the continuous use of all CPUs available. The spreadout to distribute work on more than a few CPUs is delayed as long as possible, without getting deadlines in danger, although deadlines require execution of the schedule on multiple CPUs under a total WCET assumption. Some CPUs, where the work shall be consolidated to, use the slack time of their schedule to take work off other CPUs so they could be left idle longer or completely shut down the whole time in an ideal case. This slack time has to be computed dynamically at every scheduling decision.
To get the idea clear, I make up another small example task set consisting of just two tasks: $\{T_1(1.6, 3, 0), T_2(1.7, 3, 0)\}$ With Race to Idle the tasks become arranged in a schedule as it is shown in Figure 4a. *Consolidate-to-Idle* has a different approach in scheduling those tasks: It tries to keep all tasks on as few CPUs as possible, as long as possible. Therefore a greater number of CPUs could be put in a deeper sleep-state to save more energy.
The worst case schedule of the same example task set cannot put all tasks on just one CPU without violating some deadlines. But especially on an architecture, where WCET and average execution time differ substantially the average execution time

of the jobs may lead to a different outcome, where all jobs would have fitted on just one CPU perfectly.

To avoid parallel execution the *Consolidate-to-Idle* scheduler delays necessary



(a) WCET planed

(b) some average outcome...

(c) ... that fits on single CPU

(d) outcome fitting a single CPU one period, needing two the other time

Figure 4: A sample task set $\{T_1(1.6, 3, 0), T_2(1.7, 3, 0)\}$ as it is (a) "Race To Idle" planed, (b) how an average instance might look like and (c) how this instance would have fitted on just one CPU. An other outcome (d) fitted on one CPU , but needed to fallback on two in the second period.

parallel execution as long as possible to benefit from less than worst case timing behaviour of the executed jobs. In the average case the schedule of the example task set looks as shown in Figure 4c. The start of the execution of T_2 could have been delayed so long on P_1 that it fits actually before its deadline on the schedule of P_0 if T_1 did not exhaust its full WCET. A signal would have woken up P_1 as a fallback to save the deadline of T_2 as depicted in the second period shown in 4d.
This main idea divides the involved processing units into two distinct types as described in the following subsection.

3.1.1 Processor modes

For *Consolidate-to-Idle* as already mentioned in the beginning of this section the *Consolidate-to-Idle* scheduler tries to consolidate on some CPUs while avoiding others at first. Thus CPUs can been seen as being in two different modes of operation: First, the CPUs that are being used for doing work as long as there is something to do, and second the CPUs that are avoided as long as possible.
I refer to the first kind as *consolidator* or *consolidating* CPU. These consolidating CPUs are always trying to get as much work done as possible while the other mode features two different states: A passive state, where they are just left idle because they are avoided and work still doesn't need to be spread out from consolidators, and an active state, where it is necessary for the former passive CPUs to compute jobs as their deadlines couldn't be guaranteed otherwise.
I will refer to them as *passives* or *passive* CPUs now, although they can be in the active mode. To call them non-consolidating is not sufficient, since that includes also the set of CPUs, which is not involved in the *Consolidate-to-Idle* scheduling at all.

3.1.2 Motivation for Consolidate-to-Idle

In contrast to *Race-to-Idle* one has to do dynamic slack computation in the running system to achieve correct scheduling of a periodic task set in *Consolidate-to-Idle*. This induces an overhead in computation and every task set for the task model in view, that can be feasibly scheduled with an *Consolidate-to-Idle* based scheduler, can also be scheduled dynamically in an *Race-to-Idle* fashion without such slack calculation overhead. The combined idle time of all concerned CPUs would add up as well. The question arises, how one could possibly benefit from an *Consolidate-to--Idle* CPU scheduler.
The main focus of this work is the hope for energy savings through deeper sleep states reached with *Consolidate-to-Idle* as insinuated in the introduction. The combined idle time of *Race-to-Idle* schedules may be the same, but every piece too short to achieve deeper sleep states. Measurements in the evaluation will focus on this desired effect.
Still other uses for an *Consolidate-to-Idle* scheduler are conceivable. As long as the avoidance of some CPU cores and possibly less wakeup timer interrupts yields some quality, *Consolidate-to-Idle* can be considered worth its overhead. In an inhomogeneous system with different CPU cores, the consolidation on the “best” core may

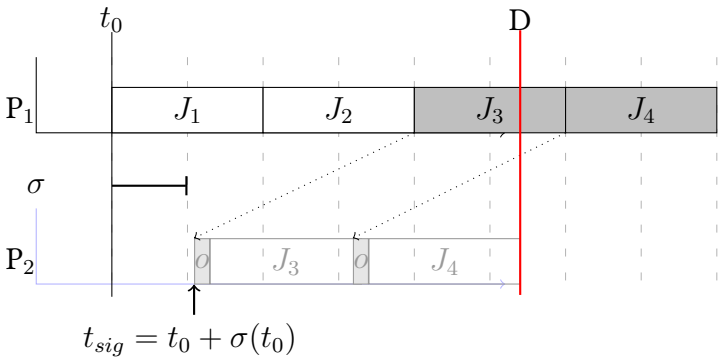


Figure 5: Pushing approach: All tasks are laid out on P_0, but the scheduling scheme allows to calculate the system slack due to zero-laxity-time consideration of tasks that have jeopardized deadlines.

provide better numerical precision, lesser unwanted electromagnetic radiation or also less energy consumption due to the inhomogeneous hardware design.
As this is totally hardware and application dependant, I will focus for the rest of this thesis only the energy economisation in a homogeneous environment.

3.1.3 Different Perspectives on Consolidate-to-Idle

For hard real-time scheduling it is important, that the deadlines of the given task set are always met. While processes are running, the *Consolidate-to-Idle* scheduler needs to decide when the currently active CPU-set needs to be enlarged, to guarantee this property.
Thus some jobs need to run on the CPUs that are woken up in the case of time pressure. The decision which job to run on the extra CPU can be seen from at least two different perspectives:
First, the pushing approach, all tasks are laid out to be scheduled on the consolidating CPU. The resulting schedule can be feasible for the average execution times of all jobs in the task-set. If due to the worst case execution time of jobs in the task set a deadline is jeopardized at least one thread has to be migrated to an idle CPU, just in the last moment in time, where this migration for parallel execution guarantees all deadlines. Some amount of time has to be considered for the migration itself and for additional cost that arises from the wakeup of the new CPU. Figure 5 depicts an example for a schedule of the pushing approach: All four jobs are laid out on the consolidator CPU P_1 although the darker shaded jobs J_3 and J_4 would miss their deadlines. The scheduler calculates the latest revision time, when those two have

to be eventually migrated. This is simply possible in this example, because all Jobs share the same deadline (marked with D) and reverse calculation of the schedule is possible as denoted by the shaded versions of J_3 and J_4. Some migration overhead hast to be taken into account.
A second view starts with a schedule, where all jobs are distributed to multiple CPUs in a way, that even if WCET is experienced all deadlines are guaranteed. But contrary to the *Race-to-Idle* approach, all CPUs delay the execution of their tasks as much as feasibly possible. They divide into two classes: The consolidating CPUs, or consolidators, and the passive CPUs. The consolidators use their system slack time to steal the jobs of passive ones, so that when average timing behavior is experienced the passive ones do not need to start work at all. I call this second view the stealing approach.

3.1.4 Task Model Refinement

To design a concrete system with one of the *Consolidate-to-Idle* perspectives shown in the Section 3.1.3 above I needed to choose a task model, which is simple enough to be understood well and rich enough to enable a *Consolidate-to-Idle* scheduler to decide when parallelization is not avoidable any longer. To make this decision, the one main requirement to the task model is, that the system slack is computable for every point in time, when such a decision on task migration has to be made. But—softening the requirement a bit—a lower bound of the system slack might be sufficient for consolidator. Such an algorithm, that never indicates more slack that actually is in the system, is called a *correct* slack computation algorithm [10, Chapter 7.5] whereas an *optimal* slack computation algorithm always finds the full amount of system slack, that is available. To effectively avoid parallel execution the best slack underestimation is harmful, but overestimation is fatal, since it might void deadline guarantees.
A fitting model that has been chosen to research the *Consolidate-to-Idle* concept is the *periodic task model* as already introduced in 2.2. To be more precise I refine the assumed model a bit: Every task consists of a serial stream of preemptible, periodic jobs. At a time a task has only one job that is ready to execute. Also the deadlines of the jobs are assumed to be *implicit*, that means equal to the spawn time of its task's next job. The deadline relative to the spawn time r_i of a job J_i is called relative deadline d_i and in our model equal to the period length of the task T_i. If a job J_i of a task T_i has already executed on a CPU, it features a runtime ξ_i greater than zero.

The slack time of a job $\sigma_i(t_c)$ at time t_c is thus given by the formula and equation 1.

$$slack = \text{time available} \quad -\text{time needed} \quad +\text{time already spent} \tag{1}$$

$$\sigma_i(t_c) = d_i - (t_c - r_i) - \sum_{d_k \leq d_i} e_k \quad -e_i^+ \quad +\xi_i \tag{2}$$

$\sum_{d_k \leq d_i} e_k$ is meant to cover the time that is needed for higher or equal priority tasks. In EDF this corresponds to task with an earlier deadline.

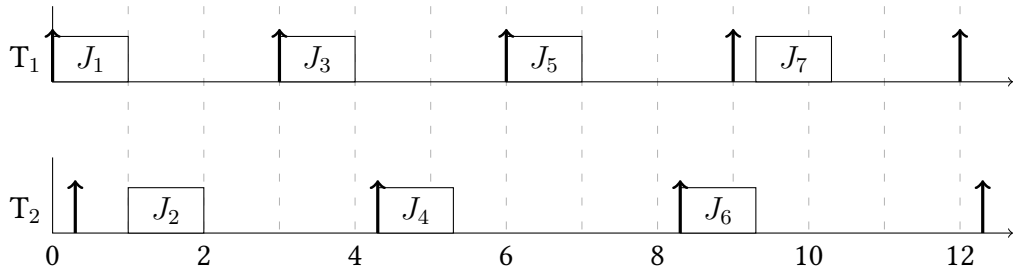


Figure 6: The sample task set $\{T_1(1,3,0), T_2(1,4,0.3)\}$.
Each task is shown on a timeline with its deadline/release time points depicted as Arrows. The jobs are depicted as blocks with their WCET as width beginning from their spawn point. The actual execution of a job has to take place somewhere between the surrounding deadline arrows. Here the jobs are laid out feasibly for uniprocessor execution. Only the first hyperperiod is shown.

As an example task set I depicted the task set $T_1(1,3,0), T_2(1,4,0.3)$ in Figure 6 along with their schedule according to a simple uniprocessor *Earliest Deadline First* (EDF) scheduler (see 2.4). The runnable job with the most urgent deadline gets assigned to its CPU as long as it has not completed or another job with an earlier deadline becomes ready.
The *Consolidate-to-Idle* paradigm is not bound to a particular method of scheduling as long as slack computation is possible.

3.1.5 Partitioned Scheduling

In [4, Section 5] Davis gives an overview of the features of partitioned multi-core scheduling. Among others Davis points out two main advantages. First, the independence of the scheduling partitions. This is good for scalability, since no global data structures are involved after task distribution and also for robustness, since

deadline misses on one core do not affect the others. Second, partitioned scheduling—after task distribution—breaks the problem down to multiple application of uniprocessor scheduling and thus a wealth of matured algorithms exist.
One disadvantage is, that the task distribution to CPUs is equivalent to the bin problem and therefore to be known as of NP-hard complexity. I will refer to the resulting partitions as runqueues—following the Linux notation of CPU associated data structures (see Paragraph in Section 2.5.2)—to avoid confusion when stealing is involved and the tasks of a runqueue gets executed on a different CPU than it was originally allocated on during distribution. Each runqueue can be modeled to some extend like a uniprocessor system, and therefore a system slack $\sigma(t_c)$ exists distinct for every such partition. As the runqueue in question will always be identifiable from the context, I will not introduce extra symbols for runqueue slack and use the system slack symbol $\sigma(t_c)$.
With the interaction of consolidators and passives, isolation between runqueues decreases. But with the limit of a static allocation of passive runqueues exactly to one supervising consolidator small groups are build, that are independent on the group level. Data structure synchronisation in such groups works for main parts of the scheduling algorithm (slack computation) without locking, since it is unambiguous who may exclusively use those structures.

3.1.6 Partitioned EDF

Partitioned-EDF is a simple scheduling algorithm and has been chosen as the basic algorithm for my scheduler. It is the extension of the simple uniprocessor EDF to multiprocessor by adding a task distribution algorithm before scheduling phase. This was chosen as basic algorithm for this thesis as it is simple but powerful enough for a variety of load and a slack calculation algorithm is known from uniprocessor EDF theory. Multiple task distribution algorithms have been proposed and feature different capabilities as the list in [Section 5.1]Davis shows. Task distribution is not the focus for this thesis and therefore completely left out in the implementation. Experiments will use manually allocated task sets.

3.1.7 Load Condition Discussion

I do not expect *Consolidate-to-Idle* to be beneficial over *Race-to-Idle* in every load situation. In a system with two cores that are loaded to 70% utilisation in the average case with tasks that could never be stolen without mandatory migration over-

head an energy consumption advantage seems unlikely. On the other hand if a task set means a utilization below an irreducible number of consolidators the application of *Consolidate-to-Idle* is superfluous as the same schedule could be feasibly achieved with static assignment to a smaller number of CPUs, without the fallback-to-parallel-execution-capabilities of *Consolidate-to-Idle*.

Of course it does no harm to have an unused fallback strategy and the consolidation can with *Consolidate-to-Idle* also be achieved, when an blundering assignment of a jobs to CPUs includes unnecessary parallel spread-out.
In any case it depends also on the discrepancy between worst- and average-case execution time, how effectively the consolidation can be.

3.2 Slack Aware Scheduling Algorithm

For the model of strictly periodic tasks, as deployed here, optimal slack algorithms exist. Therefore a slack based scheduling algorithm can use the full system slack, thats available on a runqueue. The algorithm used to examine *Consolidate-to-Idle* is an extension to the *Aperiodic Dynamic* ($\mathcal{AD}$) algorithm described by Tia in [15, Chapter 4].

3.2.1 Basic Scheduling Algorithm

The construction of a stealing approach *Consolidate-to-Idle* scheduler follows a simple principle: One needs to find an existing multiprocessor scheduling algorithm where system slack is computable and then adds the stealing component: Assign consolidators and non-consolidator CPUs before the system gets active and starts off with the consolidator using its system slack for stealing. The non-consolidators need to be activated when their system slack is exhausted. As they shall be allowed to sleep and their system slack is subject of change the recalculation should be done by the consolidator. Each passive CPU therefore needs one related consolidator that watches slack exhaustion on this passive. If the slack on a passive CPU is exhausted, it possibly needs to be activated by the consolidator—unless the most urgent task of this CPU is already executing on an consolidator.

3.2.2 Slack computation

The applied slack computation algorithm is the adaption of an optimal uniprocessor slack computation algorithm proposed by [15, Section 4] for a preemptible task set

of strictly periodic tasks. I describe it in a wrapped up form with no examples here, as it is explained with graphics and detailed explanations there and also with the same symbolic notation as used here in [10].
I take the liberty to do so, as the way of slack computation itself is not further important for the exploration of *Consolidate-to-Idle*.

For simplification I assume—besides strictly periodic preemptible tasks—also, that the first hyperperiod H over all task periods, which is the least common multiple of all task periods, must also be the end of a busy interval. That means that no phase offset pollution of a task influences with its uncompleted last-hyperperiod-job the runnqueues slack calculation in the current hyperperiod. I enforce this by the assumption, that the beginning of all first periods start at the same point in time, with an offset of zero. The slack computation is therefore only dependent on the system history up to the last hyperperiod starting point. How this restriction could be overcome is described in a given addition in [10, Section 7.5.2]. I will not add this in my implementation yet, as the postulation of a conjointly starting hyperperiod is a demand, that can easily provided in my synthetic test load. The qualitative results are the same with this restriction and with the mentioned addition could be overcome.
In general the system slack of a runqueue is the minimum of the slack of all jobs allocated on the runqueue for the current hyperperiod that have not completed yet. The slack of a particular job is given by Equation 1 as stated there. The straightforward brute-force search would thus have a time complexity class of $O(N)$, with N denoting the number of jobs in a hyperperiod. As slack computation is a frequent operation for my scheduling algorithm, it pays off to use the optimization that Tia proposed for this task model, which allows for online slack calculation in the time complexity class $O(n)$, where n denotes the number of tasks in the runqueue.
This calculation comes with the cost for the needed precomputed table of initial slack, that has a memory footprint of $O(N^2)$ in size.
For the following explanation all jobs are indexed with increasing index according to their deadline order in the hyperperiod from 1 to N.

Precomputed Slack Table To build the table the initial slack of all N jobs $J_1, J_2, ..., J_N$ is calculated:

$$\sigma_i(0) = d_i - \sum_{d_k \leq d_i} e_k^+ \tag{3}$$

Each cell of the table $\omega(j,k)$ is now filled with the minimum of those initial slacks of the jobs whose deadlines are in the range of $[d_j, d_k]$. The inital slacks of the jobs $J_1, J_2, ..., J_N$ thus form the principal diagonal of ω. At the position $\omega(1,N)$ is the system slack at the beginning of the hyperperiod and values below principal diagonal $\omega(j,k), k > j$ are empty.

Dynamic Values Besides the table ω the scheduler needs to maintain 3 more values per runqueue:

1. the total idle time I, representing the time where no task of this runqueue has been run and no stolen task was executed on the assigned CPU
2. the stolen time ST, which is the time the assigned processor run tasks from a different runqueue
3. the runtime ξ_i of not completed portions of each periodic job J_i in the current hyperperiod

Each jobs' slack, that has a deadline after t_c, has a slack at t_c of

$$\sigma_i(t_c) = \sigma_i(0) - I - ST - \sum_{d_i < d_k} \xi_k. \tag{4}$$

Job subsets Only one job per task is current and the speedup against the brute-force search in finding the lowest job slack of concern lies in the fact that one can partition all the periodic jobs according to deadlines of current jobs. Therefore I rename all n current jobs at the moment $J_{c_1}, J_{c_2}, ..., J_{c_n}$ with ascending deadlines $d_{c_1}, d_{c_2}, ..., d_{c_n}$. Each current job J_{c_i} leads to an assigned job subset Z_i that contains all the jobs in the current hyperperiod whose deadlines are equal to or larger than d_{c_i} but smaller than the deadline of the next current job d_{c_i+1} . In an other notation: All jobs having a deadline from the interval $[d_{c_i}, d_{c_{i+1}})$. The last current index in this intervall—it might be c_i for subsets containing only their single job J_{c_i}—is also

known as written $c_{i+1} - 1$

Bringing the formula for the jobs slack in, it turns out, that the slack of every job in each subset Z_i is equal to its initial slack minus the same amount as all the jobs in the subset so that the job with the minimal initial slack also has the smallest current slack of the subset. Those minimal initial slack are just the values, that have been precomputed in ω and thus the minimum current slack of all jobs in Z_i is

$$\omega_i(t_c) = \omega(c_i, c_{i+1} - 1) - I - ST - \sum_{k=i+1}^{n} \xi_{c_k} \tag{5}$$

for $i = 1, 2, ..., n - 1$, and

$$\omega_n(t_c) = \omega(c_n, N) - I - ST \tag{6}$$

The system slack on the runqueue $\sigma(t_c)$ is then given by

$$\sigma(t_c) = \min_{1 \leq i \leq n} \omega_i(t_c) \tag{7}$$

Small adaptions: The case, that the "current" hyperperiod has not been mentioned yet, but is of practical relevance. This can be the case after all jobs have completed or after fresh system initialization. System slack derives trivially from the timespan that is left to the "current" hyperperiod plus the initial system slack:

$$\sigma(t_c) = \omega(1, N) + (-t_c) \qquad |t_c \leq 0 \tag{8}$$

3.2.3 Dynamic slack reclaim

This slack computation algorithm as described here is so far only correct for the given task model, since the difference between worst-case execution time and experienced execution time has not been accounted for among the already completed jobs of the current hyperperiod. With [15, Section 4.5.1] proposes an extension of the algorithm that accounts for this unclaimed slack: The scheduler has to keep track of the differences Δe_i of all execution times against their WCETs e_i^+:

$$\Delta e_i = e_i^+ - e_i \tag{9}$$

The cumulative sum of $\mathcal{U} = \Delta e_i \forall i, d_i < t_c$ can be added to the slack of all Z_i. Additionally for each Z_i the sum of runtime difference for jobs with a deadline greater than t_c but smaller than d_i can be added.
This can be done in a way, that slack of Z_i sections still can be computed together.

3.2.4 A Progress Model: Worst Case Grains

To optimize consolidation on the consolidator cores it is important to use all time possible to avoid parallel execution. While a job is executed on the consolidator CPU the scheduler might have to decide to migrate the job to a passive CPU to achieve the integrity of all deadlines. During the execution of a job, progress is made, and the time that has to be reserved for the completion of this jobs on one execution unit decreases with the progress made. This rest execution demand e_i^* for any Job J_i is at least the already given CPU time ξ_i smaller than the WCET of J_i

$$e_i^* \leq e_i^+ - \xi_i \tag{10}$$

with its maximum the worst rest execution demand of

$$e_i^{*^+} = e_i^+ - \xi_i. \tag{11}$$

.

To get more system slack out of the already processed code, a measurement of progress done in the execution of J_i might be helpful and provide more information about how much smaller e_i^* is than $e_i^+ - \xi_i$.
For this reason I assume for my applied task model, that tasks are dividable in $k \in \mathbb{N}$ sub-parts, that have their own sub-part worst case execution times (subWCET) $se_{i,1}^+ \dots se_{i,k}^+$. At least for some tasks, it might be reasonable that there is an independent subdivision of J_i where the subWCET are independent from each other. The WCET of J_i is simply the sum of all subWECT:

$$e_i^+ = \sum_{j=1}^{k} se_{i,j}^+ \tag{12}$$

In such an independent case the rest WCET, after p sub-parts have completed, can therefore be assumed to be smaller than the sum of the remaining

$$e_i^* \leq \sum_{j=p}^{k} se_{i,j}^+. \tag{13}$$

For all already completed sub-parts execution times have been observed $se_{i,1} \ldots se_{i,p}$ below their subWCET. So the extra system slack $\sigma_i^x(t_p)$ at the point t_p after p completed sub-parts is

$$\sigma_i^x(t_p) = \sum_{j=1}^{p} (se_{i,j}^+ - se_{i,j}). \tag{14}$$

I call the smallest possible independent parts *worst case grains*. $\sigma_i^x(t_p)$ is the extra slack, that is known to a system with such grains after p executed grains. With extra slack the decision of job migration can be delayed further and a parallel execution cases avoided. For jobs, where such an assumption about independent and granular progress can not be made, it is difficult to profit from progress observation since observed progress does not provide any guarantee for shorter than worst case execution time. So e_i^* still has to be considered to be $e_i^{*^+}$.

3.2.5 Dealing With Small Time Differences

To test if some runqueues slack is exhausted, I can compute the momentary slack-time and test if this is zero or below zero. This condition has to be checked for more than one runqueue and in practice take time by itself. To take this time into account slack is in practice to be considered exhausted if it falls below a certain threshold value ϵ_Δ. Thus the condition — a runqueue k has exhausted its slack at time t — can be formalized as:

$$\sigma(rq_k, t) \leq \epsilon_\Delta \tag{15}$$

An appropriate value for ϵ_Δ is implementation specific and has to be found by measurements of the scheduler induced time overhead.

3.2.6 Consolidate-to-Idle Algorithm

Here I will describe the full interaction of the stealing-approach algorithm for *Consolidate-to-Idle* that I designed. I begin with the description of a setup phase:

Set up phase The setup phase starts by putting up runqueues with the following steps:

1. Assign a runqueue to each availiable CPU in the system.
2. Divide runqueues into conosolidators and passive runqueues.
3. Set up groups with one consolidator and one or more associated passives.

Now all tasks can be laid out on the runqueues. One may apply different strategies for this as long as they form a feasible partitioned EDF schedule. For now the tasks on one runqueue should not have any phase offset to each other.

Precomputation Of Slack Table Prior to the start of the first hyperperiod the initial slack minima of all tasks on a runqueue $\omega(j, k)$ have to be computed. See the Paragraph in 3.2.2 above.

Slack computation at time t_c for each runqueue k This has to be performed on each slack calculation for a runqueue k.

- Take the sorted list of current jobs J_{c_i} for $i = 1, 2, ..., n$ on runque rq_k .
- walk through all partitions Z_i of the current hyperperiod and get their $\omega_i(t_c)$.
- $\sigma(t_c)$ for k is the minimum amongst all those.

Consolidator scheduling action On job release, job completion or when slack is exhausted on one of all involved assigned runqueues the consolidators scheduling action takes place:

1. update $\sigma(t_c)$ on consolidator and associated passives
2. **if** (consolidator has enough slack) -> try to steal a job
   else (pick most urgent from own queue)
3. setup events signaling for upcoming events

4. wake up passives with exhausted slack, if their next job has not been stolen already

5. **if** picked or stolen job -> return executing this
   else return to idle task

The job stealing picks the first current job of a passive runqueue and tries all passives of the group until a job is found. If no job is found by stealing from the passive runqueues then a pick from the own queue stays as last resort. As a variant, a job is only stolen, if it fits the consolidators slack.
If no job is found after all the consolidator can decide to sleep and returns an idle task, that will bring up power saving modes.
In any case the consolidator has to set up a timer for the signaling of the next scheduling event. It has to find from the following possible events the nearest occurring and sets up the timer accordingly.

- slack exhaustion on consolidator runqueue
- slack exhaustion on passives, ignoring a passive from which a stolen job is already picked to run next on the consolidator-CPU
- future job releases
- the beginning of the current hyperperiod, if this is in the future

Passives scheduling action The passives shall be kept in the powersaving idle mode as long as possible. Therefore they do not react or even get scheduling events other than a wakeup call from the assigned consolidator. In this case they are low on slack and need to switch to *active mode* (see 3.1.1). The assigned now active CPU schedules according to normal EDF now. After each scheduling event the actual slack is recomputed and if the runqueue features enough slack again it *repassivates* and is under the control of the consolidator and its CPU again.

Hyperperiod housekeeping on consolidator The hyperperiod end on a runqueue must be detected by the consolidator. When this takes place the consolidator has to do extra work:
At the beginning of each hyperperiod it has to be assured that the counters I, ST and $\xi_i | i = 1, 2, \ldots, N$ counter get reseted to not let the last hyperperiod influence

the slack values of the current. This can actually be done after the last job of a hyperperiod has completed as long as it is taken care, that the I, ST and $\xi_i | i = 1, 2, \ldots, N$ stay 0 until the beginning of the new hyperperiod.

Time accounting Another important action on the runqueues is the proper accounting for time. Prior to the processing of scheduling events the time that has been spent since last accounting must be accounted to the proper one of the timecounters I, ST and ξ_i covering idle-, steal- and job-run-time. The execution of stolen job—named J_i on its original runqueue—is the most interesting case here. Their runtime has to be accounted once on the passive runqueue, where the job was stolen from, in the ξ_i counter and also on the consolidator as stealtime in ST. This is not special to the stealcase, as actually all time gets accounted once for each runqueue.

3.3 WCET Estimation

The task model of the applied design states a priori known WCETs for all periodic jobs. For practical implementation an exactly determined WCET has to be considered an illusive simplification in the model. Even a well informed estimation through static analysis of the application binary is complicated by the complexity of the platform: Code paths of operating systems with all used drivers and code libraries are highly complex and state full. And not only the operating system, also the hardware ist state full. The exact position of the hard disk's write-read head depends on the track, that had to be read before. Where a block is saved depends on the magnetic properties of the physical disk, since erroneous sectors might have been replaced by spare sectors. Those physical properties may depend on production fluctuations or environmental conditions during the whole past of the disk since then.
This is just one example deduction to illustrate that it is practically impossible to overview all relevant state of a x86_64 computer system, that might affect the actual execution time of a job. Mezzetti et. al. show effects of instruction caches on WCET behaviour [13] and make proposals to minimize them that are not always applicable for a given system.
A common approach for WCET estimation is therefore based on a simple heuristic. The completion time of the task is observed many times while the system is put in different probable load situations that might occur and affect the tasks runtime

behaviour due to caching effects for example. The maximum of all observed actual execution times is then impinged by some security factor f_s. As a starting point I assume an security factor of $f_s = 1.2$. To take static migration overhead and wakeup cost into account also constant overhead component should be taken into account and compensated by an additive margin m_s. This margin includes overhead compensation for one complete cache trashing overhead due to migration of a running job, and for wakeup cost for an idle CPU actually twice.
The scenario, where worst-case-wakeup-time t^+_{up} can occur twice, but must have been considered only once in the WCET is the following: An idle consolidator wakes (first wakeup) due to the spawning of a Job on a passive supervised by this consolidator. The job is stolen by the consolidator. If the jobs slack on the passive would be near zero the passive would be woken up then, but as it is for the stolen job, the consolidator ignores this as wakeup reason for the passive. But then the consolidator native jobs exhaust consolidator slack. The consolidator drops the job and picks the urgent job from its own queue. If the progress the passive's job has made on the consolidator is not considerable, the passive has to be woken up now (second wakeup). So WCET e^+ has to include t^+_{up} twice.
A different way to deal with this scenario is simpler: Consolidators steal only jobs, when their own slack is large enough to compensate with stealing time for the t^+_{up} of the passive. When the consolidator calculated the slack for its job queue its obviously actually running, therefore not in a sleep state. If thus its slack is larger than one t^+_{up} of the passive, it may steal passives job and the second t^+_{up} is compensated sufficiently.
The other part is the cache coldness overhead due to extra-migration. The maximum of measured execution times of a job J_i (empirically measured in j, repetitions under changing load conditions) $max(e_{i,j})_{measured}$ should contain compensation for one start on a cache cold cpu $t^+_{migrate}$. But the scenario above shows, that after being dropped from consolidator an other start on passive core occurs.
Whereas t^+_{up} should be the same for all jobs, $t^+_{migrate}$ depends on the cache memory footprint of a job and is therefore to be estimated distinctively for each job. This can be done by static cache footprint consideration, or—as the estimation of WCET is done—heuristically. Thus the final considered WCET for a Job J_i results in

$$\begin{aligned} m_s &\geq (t^+_{up} + t^+_{migrate}) \\ e^+_i &= m_s + (f_s * max(e_{i,j})_{measured}), \forall j \end{aligned} \tag{16}$$

4 Implementation

To reflect upon *Consolidate-to-Idle* purely in the synthetic contemplation of a model gives—in the best case—an incomplete view. In the following section I will give some details on the implementation I made to evaluate the concept *Consolidate-to--Idle* in a real world operating system.
To achieve the integration, several parts of the design need to be mapped to the targeted system. Although all implementation parts are interlinked with each other, I will discuss seven aspects of them distinctly in this section:

First, I will explain the general interface I built for the communication of the task set in the userland and the scheduling Linux kernelmode code on the other so that a deadline aware task model can be instantiated. Second, I will take a deeper look on the implementation details of the EDF and slack computation algorithm in my scheduling class. The following three subsection show several aspects of timing concerns: Those are with a subsection each the *high resolution* timers from the kernel that are used for signaling in my implementation, a general review of the used clocksources and a discussion of practical worst-case-timing estimation, which is fundamental for the real-time model application. The last two subsections target practical energy consumption concerns: First, the measurement tools I set up with task model conformig exploitation of hardware provided measurement features and second, the Linux provided way to actually save energy and some pitfalls with its application.

4.1 Linux Integration Design

While the scheduling paradigm introduced in this thesis should be applicable to a variety of operating systems, for several reasons the decision was made to integrate the first test-scheduler into the Linux kernel:
First, Linux provides a sophisticated multi-processor enabled thread architecture. It already features functions for thread migration that can be used to perform load balancing or CPU hot-plugging for example.
Also included are drivers and abstractions for frequency scaling and energy saving power states. These are well testet and implemented for the majority of the most commont x86_64 architecture processors—not only by a worldwide user community, but also by hardware vendors like the processor manufactures of the used test hardware, that contribute to those energy consumption improvement drivers.

The huge spread compared to research oriented or other less used operating system kernels also brings the benefit of a large amount of already available software in the Linux userland. This facilitates the evaluation of *Consolidate-to-Idle*s features. Finally, one of the pivotal reasons for an implementation in Linux was the prominence of Linux in the scientific community. It is easier for other researchers to interpret and compare results form a widespread kernel, than from special kernels. This is especially important since—to the best of my knowledge—this scheduling paradigm has not been implemented and explored in any publication before.

Those benefits of the Linux platform come with the following drawbacks: The Linux scheduler is a complex structure. A present version of Linux features over 13,000 total physical source lines of code only for the scheduling and task management related code[2]. This code is constantly subjected to changes; and due to only little available documentation a fair amount of research overhead is induced that is required to understand all the existing concepts in the kernel.
Additionally Linux completely lacks the concept of timing constrained thread scheduling. Neither periodic jobs in a task nor deadlines in general are featured in the Linux task model. Normally this is not a problem, since Linux is mostly used in best-effort manner and more concerned with the fair distribution of restricted resources rather than jobs' exact placement in time. Even the two scheduling classes provided by the *RT* scheduler module are not deadline aware, but their simple design and their high priority in the scheduler hierarchy make it possible for a careful system designer to build some timing constrained behaviour in the userland without kernel enforcement of timing.

4.1.1 Systemcall Interface

I named the research implementation "*Universal GLobal Yield* scheduler (UGLY scheduler)" because the original idea was to use the already existing and scheduling related systemcall `sys_sched_yield()` for the communication of user-mode application and scheduling class in the kernel. This approach turned out to be impractical since `yield` is sure of little use with its original semantics for the applied task model, but on the other hand defined in the core scheduler with a

[2] This number is the output of sloccount tool running against the vanilla kernel archive on my computer in version 3.8.Counted were all files under kernel/sched/ and aditionally include/linux/sched.h. This number gives only a clue to codesize and should not be mistaken as an exact complexity measurement

certain behavior. Parameters like WCET or deadline can not be transported by a yield without invasive modifications to the Linux scheduling and task model core. The next candidate for a communication interface was the adaption of the `sched_setparam()` call, that seemed to be fitting much better for the use case of setting scheduling parameters. It turned out, that using this interface was not usefull either for two reasons: First the use of a modified `struct sched_param` turned out to break the ABI for even normally scheduled threads. The second was the behavior of the normal `sched_setscheduler` systemcall, that is in unmodified Linux responsible for changing the scheduling class of a thread: This systemcall leads the core to invoke the enqueue callback of the new scheduling class, before the scheduling parameters are made know to this class. While this is sufficient with the usual task model of the Linux kernel it is better to know parameters like deadline and WCET prior to queueing the task in the runqueue of the scheduling class.

All this lead to the insight that a private systemcall for the UGLY scheduler lead to a cleaner interface, more flexibility for experiments and even better ABI compatibility to unmodified user mode tasks, that are not put on the UGLY scheduler. Alongside the systemcall interface I made a userland programm, that enabled me to trigger all scheduler providet functions

Listing 1: Help page excerpt for usermode scheduler control command

```
available commands::

 Mode: reinit
  issue reinit command
  -r, --reinit               issue reinit command
  -i, --homecpu=INT          The cpu that is going to be
     consolidator

 Mode: rtistart
  race to idle mode start
  -R, --rti                  start race to idle mode

 Mode: spawntime
  set global spawntime
  -S, --setspawn=LONGDOUBLE  The timeoffset from now for first
     spawnline

```

```
16 Mode: timeprint
17  show actual scheduler time
18  -t, --gettime              get kernel time
19
20 Mode: cpuinfo
21  print infos about one rq
22  -I, --infocpu=INT          Ask the scheduler for information
       about one cpu
23
24 Mode: hypercalc
25  recalculate slack table
26  -H, --hyperrecalc=INT      Ask for recalc of slacktable on given
       cpu
```

4.1.2 Preemptive Periodic Taskmodel In Linux

As described in Section 2.5.1, the Linux kernel offers a scheduling system with multiple classes. And to add scheduling policy is naturally done by adding such a class to set of available ones. A custom policy in combination with the systemcall configuration described above (4.1.1) forms the main part of the representation of the periodic taskmodel (see 3.1.4) and its scheduling policy. As the mapping is not just a strait scheduling policy class there are some peculiarities though:
The special UGLY class gets responsible for all task with the periodic task model described in 3.1.4. As the periodic stream of jobs that a task consist of never has jobs in parallel, I mapped the Task directly to one linux thread that is assigned to the UGLY class. After each jobs completion the application uses the systemcall interface to tell the scheduler to respawn the next job of the task in the next period of this task.
The waiting for being picked again is not implemented by Linux typical blocking with a full dequeue. The reason for that is that the scheduling cores behaviour against the scheduling class would either be indistinguishable from a final disappearance or at least awake the need to hold state for each task connected to a runqueue and major changes to the core scheduler. To put a task in full control by the UGLY class without changes to the core and because the task model used for the algorithm depends on nonblocking jobs I decided to prohibit blocking in the Linux way and implemented the waiting for the tasks next active job instance by just not picking it in the scheduling class. From the Linux standard semantics point of view the thread, that represents such a task stays runnable, on runqueue. This means

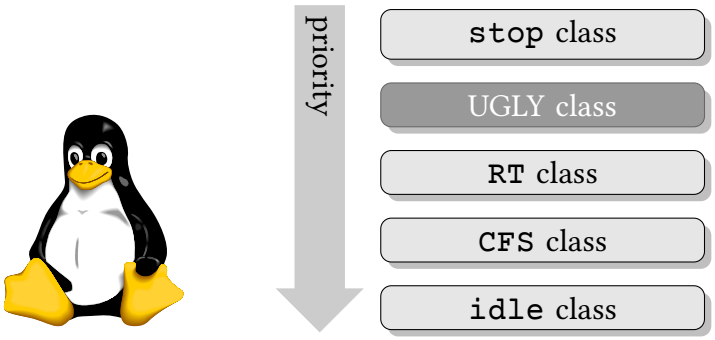


Figure 7: Hierarchie of scheduling policy classes in the Linux kernel from highest to lowest. Highlighted is the rank of the non standard *UGLY*-class, that is subject of this thesis' implementation.

I have to do all possible blocking work of the thread before its put into the UGLY class and working as the periodic thread. Even memory allocations in the kernel itself can block the thread, on which behalf the kernel executes. To avoid blocking in my implementation all tasks and the kernel scheduler's management structures allocate the memory needed prior to the phase of periodic job execution.
There are more implications of this nonblocking assumption: The scheduling class can be left without or minimal stub implementations of some of the core scheduler callbacks: `check_preempt_curr` for example is not necessary, since a running job on UGLY would imply that all other currently existing UGLY task can expected to be ready at the time already.
Another speciality of threads in the UGLY class is that the normal interface to change the scheduling class of a thread can not be used to bring them into the UGLY class for the first time. The core would enqueue them before the callback of class arrival would be called and at the time of enqueue I want to know all necessary parameters of a task in the model to enqueue them sorted in the runqueue. I set up a systemcall command for transition to UGLY instead and prior to this the tisk is responsible to tell the UGLY class via another such command its parameters WCET,period length and a phase offset against the global startline. No thread is picked by the UGLY scheduler before this global start time, which gives the possibility to set up all tasks from threads while they are still on other scheduling classes of lesser priority in the hierarchy. As depicted in Figure 3 I put the UGLY class on top of the RT class.

This is not of special importance, because for energy conserving tests a CPU with UGLY tasks running should no have other threads left from other classes[3]. It adds some flexibility to run threads with lower priority instead of the idle task otherwise, but this has not been explored in this thesis.

4.2 Task Representation

I represent each task in the system with a Linux process, that is a thread with an exclusive address space. For scheduling purposes I need to keep track of all threads that are assigned to my scheduling class. When a thread enters my scheduler—by using the systemcall interface—all important information for the task it represents is already annotated in the `task_struct` of the thread.
On the per CPU runqueue structure I have a list structure where I can link all thread structures that are given to a runque. I insert each thread when the `enqueue_task` callback of my scheduler gets called in a by deadline sorted fashion. On job completion the `task_struct` gets reinserted into the list, that my list always represents a doubly linked list of jobs, that are either current or not spawned yet.

Listing 2: Commented substructure used for each ugly scheduled task

```
1  struct sched_ugly_entity {
2    struct list_head edflist;   // membership in sorted uglytaskring
       in rq
3    ktime_t spawntime;          // calculated next spawntime
4    ktime_t phaseoffset;        // period offset against hp
5    ktime_t wcet;               // wcet specified
6    ktime_t deadline;       // actual absolute deadline
7    ktime_t lastpoint;          // legacy task model used 0
       laxitypoint
8    ktime_t period;             // = relative deadline
9    struct rq *orig_rq;         // link to original rq
10   ktime_t ksi;                //runtime recived in current period
11   unsigned int periods_in_hp; // hyperperiod / period
12   u32 * hyperidx;             // small array of indices in
       hyperlist
13   u32 hyperinstance;          // index in hyperidx array for each
       se
14   u32 hyperid;                // identyfies hyperpeiriod
```

[3] Of course the idle task, on its own class IDLE, could be seen as the only exception here. But strictly speaking is the idle task not represented by a full blown thread in Linux

```
15   unsigned int generation;     // counts jobs in task
16   int on_rq;               // flag for uglyrq membership
17 };
```

4.3 Edf and Slack Computation Implementation

Deadline aware real-time scheduling was only one part of the challenge to cope with in the implementation of the research test scheduler. Since a fundamental requirement for the *Consolidate-to-Idle* principle is the ability for the scheduler to calculate the actual amount of system slack the algorithm of Tiu [15, chapter 4] as described above in section 3.2.2 was subject of implementation in the Linux kernel. As stated in the design section the slack computation according to the algorithm listed in Section 3.2.2 is divided into two stages: The first stage is the precomputation of the initial slack values, while the second stage is the repeated actual slack calculation during the active phase of scheduling.
As hyperperiods allow for many jobs per task in such a hyperperiod, my doubly linked list is not sufficient to do the calculation easily. Thus on the initial slack table pre-computation I put up an other datastructure. This list of all jobs in a hyperperiod and orders them by deadline. Then each scheduling entity gets a pointer to this list, where it can pick all hyperperiod wide job indices for its task. These are needed in slack computation to account for jobs, that are currently not in the real joblist, since their predecessor still has not completed.

4.3.1 Precomputed Slack Table

The precomputed slack table $\omega(j, k)$ (see has for both of its indices a possible value range from 1 to N. Without limitations for the use in the slack computation, i could always be assumed to be less or equal than j so that $\frac{N*(N-1)}{2}$ entries are unused. Therefore I have left out unused values for the linear memory representation of the table. The table indices i, j relate to the array index r in the following manner:

$$r(i,j) \equiv N(i-1) + j - 1 - \left\lfloor \frac{i(i-1)}{2} \right\rfloor \tag{17}$$

4.3.2 Handling Transient Overload

Transient overload is not handeled by this implementation. One of the disadvantages of EDF is its unstable behavior if some inevitable overload situation occurs [10, Section 6.2]. In such a situation EDF for soft realtime system an EDF scheduler needs to be accompanied with a recovery strategy, that sets the system back into stable behaviour. The scenario for this thesis is oriented at hard real-time requirements. Also is application specific knowledge is needed, to decide which guarantees shall be void in a way, that compromises the system quality the least.
For simplicity this has been neglected in this thesis: Any deadline-miss is considered to be failure and the only application specific knowledge that is valid for fully synthetic hard-real-time workload. To address small jitter induced overload deadline misses the WCETs of all tasks should include some buffer. The security factor on WCET of all jobs gives to some extent an instrument for avoiding overload by having a bigger safety margin.

4.4 Clock Considerations

Linux standard kernel timekeeping architecture offers various clocks for dealing with time resources. The `CLOCK_MONOTONIC_RAW` offers 64 bit wide 1 ns grained time. It is monotonic and not subject off any time adjustments by userland programs. Still its a systemwide clock. Theoretically—but for now highly unlikely—a CPU-local clock could lead to problems, when passives gets activated, recalculate their slack and come to the conclusion that they have enough slack to sleep again in contrast to the result of the consolidator. I therefore always let actives complete a task, as the consolidator would not have woken them without slack urgency. On systems with only CPU local clocks this would make the implementation robust against small amounts of inter CPU clock drift.

4.5 High Resolution Timer Interface

For the signaling of oncoming scheduling events like an impending exhaustion of a passives runqueue slack I needed to utilize one of the timing interfaces provided by the Linux kernel. `hrtimer` is the most advanced timer interface that linux provides. It features signaling of points in time that can be specified by values modern nanosecond based time values.

4.6 WCET: Practical Estimation

As stated above (see 3.3), I would like to estimate the WCET of a task by repeatedly observing the execution time and impinge it with safety margin.

My synthetic benchmark jobs monitor any deadline violation to see if this estimation is good enough to cover all occurring worst cases and also the *Consolidate--to-Idle* induced migration overhead. This simple approach may be either too pessimistic or dependent on the tasks complexity also too optimistic for some workload. With the tunable security factor I assume this to be fair enough, since overestimation of the WCET may influence the application performance, but not the study of *Consolidate-to-Idle* qualitatively.
I choose a quite simple strategy for practical measurements: Put into a verbose mode, my test-tasks account their runtime and print it out on the standard-output when a new maximum is reached. Accounting itself and the printout is not taken into account.
This is deactivated in normal (energy measuring) mode, as the accounting and even more the printing would increase the full WCET of this task.

4.7 Energy Overhead Measurement

For comparison of different scheduling strategies one depends on realistic approximations for all characteristics of the scheduling process. As stated in Section 3.1.2 energy consumption is a number one concern for this *Consolidate-to-Idle* application and has to be measured somehow for success control.
Since energy aware computation gets more interest, processor vendors started recently to build in facilities in their products, that enable energy consumption measurements from within the software.
In their contemporary desktop processor products Intel started to provide model specific CPU registers, that indicate the amount of energy used in different power domains, since last overflow of that counter. These so called *running average power limit (RAPL)* counters can therefore be used, to quantify energy consumption, when sampled not after a second overflow of their registers. This is hardly a problem for my measurements, since an overflow is not to be expected twice in some hours. Hähnel, Döbel, Völp and Härtig have shown in [6] how to improve energy measurements code paths that are short compared to the 1 ms update interval of the counters. As I head to compare test runs of at least 1 s, I have no need to apply their findings for my measurements.

Recently Hackenberg, Ilsche, Schöne, Molka and Schmidt pointed out, that RAPL counter are not physical energy measurements but rather based on an calibrated model of energy consumption.[6] Thus they compared the measurements done with RAPL to electrophysical measurements and found systematic variation measuring various workload. The advice not to trust in the absolute values up to the finest graduation they provide
In my measurements I want to compare values of similar workload of the same machine. Model based counters as RAPL should be sufficiently accurate for this case. Their availability in off the shelf hardware make them the best choice for the practical evaluation of *Consolidate-to-Idle*.

4.7.1 Joesw Implementation

The RAPL counters are implemented to be interfaced as model specific registers (MSR) that can only be accessed from privileged running code, thus from the kernelspace in Linux. For easy measurement—controlled by user land applications—I designed a kernel module, that reads the time and the energy counting MSRs and provides a stopwatch like interface to the userland, leading to the name *Johannes' own energy stopwatch (Joesw)*.
Implemented are the the numbers of RAPL concerned registers for SandyBridge and IvyBridge microarchitecture models by Intel listed in [8]. Therefor and can be used on machines with either of those processor familys.
Udev, a system for creation of special device files that are interfaces to kernelspace drivers in Linux, is capable to create dynamically allocated device files for Joesw, since the kernelspace part registers for the private device class Joesw and udev rule files are supplied, that trigger the character device file on which some ioctrl functions are specified. Those functions enable the user to start, stop, reset and read out the stopwatch. A simple helper program is provided, that wraps the ioctrl interface to an interface callable from the system shell. This provides the user with an easy script controllable interface, easy to use from userland scripting.

4.8 Idling For Profit

To achieve actual savings in power consumption the idle passive CPUs needs to be put in power saving states. The implementation for this is already build into the Linux kernel, which was one serious reason to take Linux as the platform for a first test implementation of a *Consolidate-to-Idle* scheduler. Pallipadi, Li, and Belay

describe in [14] their effort to build a unified infrastructure for power saving during idle state in Linux. This lead to the *cpuidle* framework, where a unified interface is provided. It interlinks different drivers concerning idle modes on different hardware platforms on one side and controlling software in the Linux kernel on the other.
One of such Linux drivers adapts to platforms covering the so called *Advanced Configuration and Power Interface (ACPI)* -standard. The principle goal of ACPI is to empower the operating system of a platform to do hardware detection and configuration and to use determine power management decisions rather than in the former standards like APM where all policy had to be implemented in the firmware of the plattform [7]. Therefore ACPI defines a complex standard with domain specific languages that allow the communication of operating system with the firmware of such a platform. Linux ACPI drivers are often not provided by the hardware vendors and inaccuracies lead to problems in full support of power management in some ACPI platforms.
As my scheduler is is not directly concerned with energy saving, but relying on the power saving of the cpuidle framework it therefore has to be checked, if the power management of the platform is sufficiently supported by the cpuidle framework.

5 Evaluation

This section discusses all matters of evaluating the *Consolidate-to-Idle* principle with the help of the research scheduler implementation. The evaluation is left in an fairly incomprehensive state since difficulties in the implementation of the system in Linux did not left as much time as I needed to understand the practical behaviour in full extend.

The Evaluations goals could be put into two main question categories:

Correctness: The first questions evaluation brings up, is the question if the scheduler works as it is expected by the model. Are the tasks really run in the way they should according to their deadlines and periods. As the taskmodell from the design is different from what is normally applied to Linux threads this question is an essential prerequisite to any further qualification. A question that leads to the second category is how correctness may be void by concessions that have been made in the implementation, if special working set parameters get applied.

Quality The second categorie is about how well the resulting performance is. And for this thesis the may performance issue is energy consumption. Therefore it is of great interest how much energy is consumed by sample task sets scheduled according to my *Consolidate-to-Idle* scheduler compared to conventional *Race-to-Idle* scheduling of the same task set.

First I am interested in how much the correctness of the model application can be void by concessions that were made in implementation. And second the question come from the initial motivation of this work: If so, how much energy saving could be achieved compared to *Race-to-Idle* schedule?

5.1 Requirements On Evaluation Task Sets

For a broad understanding of the properties of the scheduler I needed to bring up a flexible framework to put different type of workloads on the scheduler.
As stated in Subsection 2.2 blocking tasks needed to be avoided at all. A blocking task would ruin the calculability of the system slack since the task would be dequeued from the scheduler by the Linux scheduling core. This is a major weakness for the direct use in real world applications of my task model implementation

in Linux. Testing the scheduler with non-synthetic, complex application task sets, like an adapted video decoding software showed off to be impracticable, since blocking is a common behaviour in various situations in the thread implementation of the Linux kernel. Even the default memory allocation scheme in privileged kernel code is not guaranteed to be non-blocking for the task on which behalf the kernel code is executed.
Therefore a special synthetic benchmarking tool was crafted, that was useful for a synthesis of different workload situations, without the risk of blocking a thread that constitutes one of the test tasks. I designed the main benchmarking task to be flexibly tuned in two main dimensions of resource consumption: On the one hand the memory footprint size on the other raw computational effort. A simple matrix multiplication task turned out to be simply tunable in both mentioned dimension. For exploration of different memory footprint sizes the matrix dimension of random test matrices is adjustable. To achieve different types of computational complexity I added a counter, that determines the number of matrix multiplications done per job. All memory needed for the computation gets allocated before the task is set up in the UGLY scheduler. This is to avoid the possibility to block the task during fulfillment of allocation requests.
This results in a synthetic benchmarking task that has the following footprint:
In terms of memory consumption one needs at least—given the parameter d—space for 3 matrices of $d \times d$ elements with their particular element size. I decided to go with integer elements that are of 32 bits in size on the testing platform. Little extra memory is needed on the stack for program flow control and accounting and also for the actually code performing the matrix multiplication. This amount is partially shared among multiple instances and therefor not really easy to quantify. For the initial test data set sizes this amount of memory was neglected.
Listing 3 shows the help page for the benchmarking task. In combination with the command interface tool to the scheduler a variety of load situations can be created as simple shell scripts that run on a CPU not concerned with *Consolidate-to-Idle* scheduling to minimize distortion.

Listing 3: Help page excerpt from testframework task

```
1 Usage: schedtest [OPTIONS]...
2
3   -h, --help               Print help and exit
4   -V, --version            Print version and exit
5   -u, --useugly            use the ugly scheduler  (default=on)
```

```
  -r, --reinits             this task starts reintialisation  (
     default=off)
  -i, --homecpu=INT         The cpu this task is to be asigned to
  -w, --wcet=LONGDOUBLE     worst case execution time (in seconds)
  -p, --period=LONGDOUBLE   period (seconds)
  -P, --phase=LONGDOUBLE    period phase offset (seconds) -> will
     influence
                              spawnpoint
  -d, --matdim=INT          dimension n matrices will be n*n in size
  -n, --mulrounds=INT       number of repetition of matmul
  -v, --verbose             increase verbosity of output  (default=
     off)
```

I created also a test task similar to the matrix multiplication one in its behaviour, but syncronizing the extraction of energy counter and system clock values instead of computation. This test task is configurable in the number of hyperperiods a measurement shall take place and controls the energy stopwatch in the kernel module (see Section 4.7.1) accordingly.

5.1.1 Testdata Characteristics

To explore multiple cache induced effects on the test system, the cache architecture has to be known. In order to get the cache dimensions on my test system, I used the information provided in by the Linux kernel about cache layout and sizes. Marcus Hähnel wrote a handy tool that I could use with permission, that parses this information and generates an overview for each cache level.

5.1.2 Testbench Setup

For testing, the Linux kernel with my scheduler code was embedded in an userland environment that has been build with the *buildroot* Linux distribution [1]. So I could compile the Linux kernel and a complete GNU/Linux userland into a initial ramdisk that can be lodet by the bootloader of my testsystem or directly in an system emulator for debugging purposes. As I packaged all test task sources with GNU autotools, they integrated with little afford into the buildroot compilation environment.

Qemu Setup A setup of a virtual machine provided a very usefull testbench for getting the code for the scheduler operational and behaving correctly. Listing 4 shows the parameters used for most timing critical debugging work, where fully virtual time is applied. Thus the even the kernel embedded CPU-scheduler can be single stepped by the virtual machine monitor Qemu in conjunction with the GNU debugger (gdb) controlled from the host machine.
Timing uncritical debugging was done with a different parameter set, that allowed the use of hardware supported virtualisation-techniques. This gives faster virtualisation to develop userland measuring applications for example.

Listing 4: Timing aware Qemu parameter set

```
#!/bin/bash
qemu-system-x86_64  -smp 3 -icount 4 -m 1024\
        -ctrl-grab \
         -s\
        -net none \
        -serial stdio \
        -kernel \$IMGPATH/bzImage \
        -append "console=ttyS0" \
        -initrd \$IMGPATH/rootfs.cpio
```

With Qemu a good part of the first question about the correctness of the model aplication was answered positively. All details like slack table calculation, hyperperiod change or the hrtimer settings can be inspected by singlestepping the code. Jobs get executed as the model expects.
But for energy measurement and to verify function on a real system real hardware platforms must be used.

First Test Machine The first Test were run on a contemporary Intel PC system with the following specifications in the first place:

1. CPU: Intel® *Sandy Bridge* microarchitecture
   Core™ i5-3550 CPU @ 3.30GHz
2. 4x 256kB L2 Cache
3. 1x 6MB L3 Cache
4. 4GB System RAM (DDR3)
5. Intel® Desktop Board DH77EB

6. Intel® H77 Express-Chipsatz

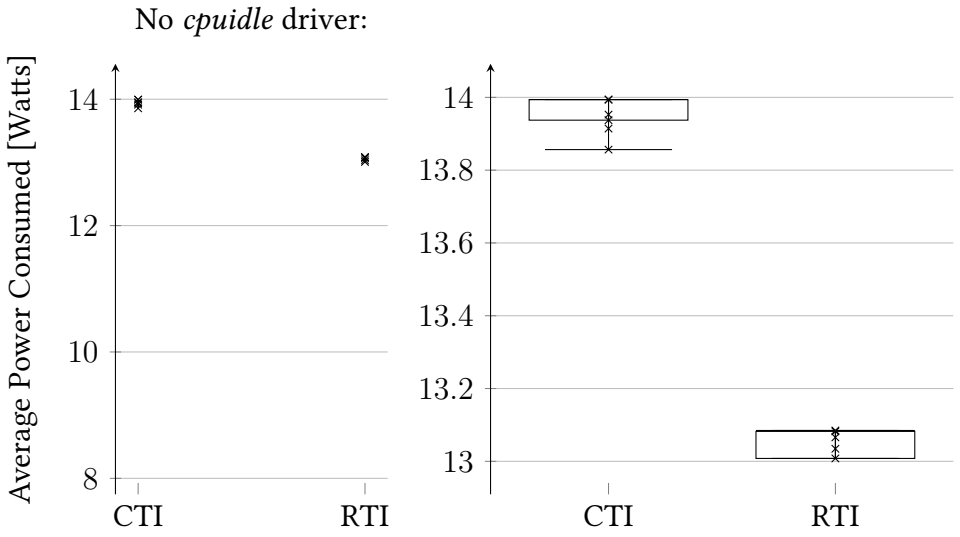
Second Test Machine Since I encountered problems that were specific to the hardware and firmware of the first test machine (see results below 5.2) I had to change to a test machine with different specifications:

1. CPU: Intel® *Sandy Bridge* microarchitecture
   Core™ i5-2400S CPU @ 2.50GHz
2. 4x 256kB L2 Cache
3. 1x 6MB L3 Cache
4. 2x4GB System RAM
5. Intel® Desktop Board DB65AL
6. Intel® H77 Express-Chipsatz

5.2 Energy Measurement Results

After correctness was achieved through iterative development cycles with tests the emulation environment, the first energy measurement results were disappointing: Measurements showed an energy consumption that were in average 6.8 % above for the *Consolidate-to-Idle* scheduler compared to the results measured using a *Race-to-Idle* scheme. All measurements that were made with the first test task set $T_A : \{P_1\{T_1(.5, 5, 0), T_2(1, 5, 0)\}; P_2\{T_3(.5, 5, 0), T_4(1, 5, 0), T_m(0.000015, 5, 0)\}\}$. Figure 8 depicts results for 5 hyperperiods at a time with energy consumption normalized to time intervall of measurement in a boxplot.
The search for a profound analysis of the negative result lead to a simple main reason: For the first test hardware no working cpuidle driver (see 4.8) was available. The idle cpu was therefore not put to sleep properly.

I suppose the excess consumption in the cti schedule comes from function units in the chip, that are powered on fully if just one core uses them but I have not been able to track this down.
The system was replaced with the second test system, where the acpi_idle driver was successfully loaded for the cpuidle framework. This resulted in a significantly lower idle consumption of 3.542 W. A comparison between the two systems is inopportune for various reasons, but a situation similar to the one on the first



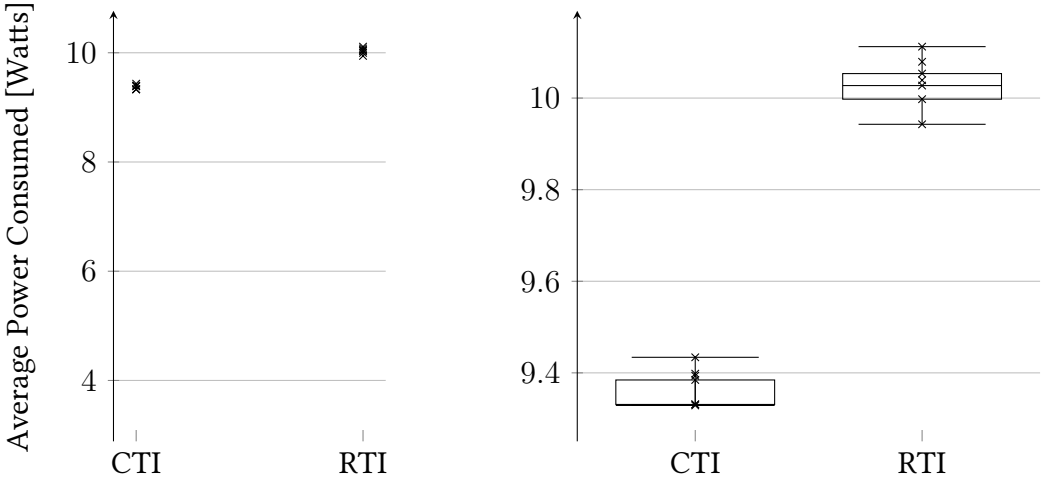
(a) Measurement Plot (b) Zoomed-In Boxplot

Figure 8: First average power consumption values for taskset T_A under *Consolidate--to-Idle* and *Race-to-Idle* schedulers. Each datapoint represents mean energy consumption over five hyperperiods. Energy values form RAPL counter units have been normalized to measurement time resulting to the unit Watts. Subfigure (a) shows all measured values with an ordinate intercept corresponding to the average total-idle power consumption of about 8.315 W. In (b) the same data is plotted as an boxplot [17] describe

system with no working cpuidle driver could be provoked on the second system by disabling the use of the C-states C2 and C3 with the kernel command line parameter `idle=halt`. The system showed an idle consumption of 8.789 W then.
I was not able to reproduce excess consumption on this machine, when `idle=halt` was given for the same taskset.

Exposed to the same T_A taskset workload as in the first test, the second system brought up different results as it is depicted in Figure 9. Here a stable power conservation was achieved. The boxplots show, that the resulting values were densly distributed and significantly differed between *Consolidate-to-Idle* and *Race-to-Idle* .

working `acpi_idle` *cpuidle* driver:



(a) Measurement Plot (b) Zoomed-In Boxplot

Figure 9: Average power consumption values for taskset T_A on second testsystem. Parameters and depiction as in Figure 8. Ordinate intercept in (a) corresponds to the total idle power average of the second testsystem at about 3.542 W. (b) depicts the same dataset as boxplot again.

5.3 Concerning Cstates

The ACPI standard organizes different power saving modes of a CPU in so called Cx states [7, Chapter 2.5]. The lowest state C0 is the state of actually executing code. C1, the second deepest sleep state is characterized by a wake-up latency, that shall no be a concern to the operating system and therefore functional units stay fully prepared for execution of code. The deeper sleepstates from C2 on are actually saving energy at the cost of having a bigger wake up time cost.

5.3.1 Cstate Parameters of Testsystem

The worst case wakeup time that should be considered by the system is handed from the firmware to the operating with the ACPI interface. The test system for my scheduler uses the `acpi_idle` driver of Linux to provide power saving idle states.

The driver interfaces the ACPI capable firmware of the platform and exports besides other things the latency, the usage number and the usage time of

every usable power state. These informations are exported to the user-land access via the *sys-filesystem* and accessible under the directory path `/sys/devices/system/cpu/cpuidle/` and specific for each CPU cpu_i of a system under `/sys/devices/system/`cpu_i`/CPU/cpuidle/`. The states can be disallowed there according to the cpuidle documentation [3], if one wants to minimize of latency of the system at the cost of energy consumption. Table 1 lists the values I found on the test system.

	state0	state1	state2	state 3
name	POLL	ACPI HLT	INTEL MWAIT 0x10	INTEL MWAIT 0x20
descr	POLL IDLE	C1	C2	C3
latency	0	1µs	80µs	104µs

Table 1: Powersaving state information provided by `acpi_idle` driver on testsystem

I did not consider deactivation of deeper states as an valuable option here, because the deepest available state reports a worst case wakeup time of just 104 µs. This is about one order smaller, than the latency induced by Linux scheduling framework interrupt reaction latency. If C3 would be forbidden, than only C2 would be left as an energy conserving state. C2 is listed with a latency advantage of 24µs. Every job execution can contain the wakeup overhead only once as jobs are considered to be nonblocking. The worst wakeup overhead is therefore simply accountable for in the constant part of the e_i^+ overhead.

Cstate Idle Time I wanted to measure the difference in time spent in the Cx-states that lead to the energy economization for the schedule of T_A. Some tools to read out Cx state files are available for Linux userland , and I tried some out to track down the influence of the scheduling paradigm to the Cx state usage on my system, but with their dependencies those programs did not work out well or did not work at all on my minimal userland.
Thus I wrote a shell script that dumps the Cx state information in a temporary file in system memory once—and once again after a fixed interval. During this time—120 seconds here—the UGLY scheduled test tasks to see how scheduling paradigm influences the sleep state usage.
For the time and usage count the outcome of a typical test run is listed in Table 2:

Given is only the difference between start and end of the 120 seconds, also only the numbers for the C3 state are listed, the deepest available state on my system, since the other values did not change during the measurement for the CPUs concerned.

	Consolidator		Passive	
C3 states...	Δcount	Δt[µs]	Δcount	Δt[µs]
CTI	2061	37,590,636	483	119,997,256
RTI	782	73,899,505	139	73,879,173

Table 2: Typical run on CTI and RTI: C3 state usage difference during 120 seconds

If the total sleep time of both cores is added up and the difference of CTI and RTI normalized I get the C3-state benefit factor $\Delta bs3$

$$\Delta bs3 = \frac{((37590636\mu s + 119997256\mu s) - (73899505\mu s + 73879173\mu s))}{(120s)} \quad (18)$$

$$= 0.081743\ . \quad (19)$$

That means that for the average second of execution of this taskset my CTI scheduler sleeps 81.743ms longer than the RTI one.
When I take the idle measurements in account which were 3.542 W with $C3$ and 8.789 W without power saving staates

$$\Delta P_{Cdiff} = (8.789\text{W} - 3.542\text{W}) * 0.081743 \qquad = 428.906\text{mW}\ . \quad (20)$$

Indicated by the Energy counters during this 120s measurement was a average power difference of about 611mW. This is a difference of about 40%

5.4 Implementation Limits and Open Questions

I expected the behaviour of my *Consolidate-to-Idle* scheduler most beneficial with task periods so small, that for a *Race-to-Idle* scheduler the slack would not have been enough to go into sleep states, while passives of *Consolidate-to-Idle* easily could. Disappointingly I was not able to get measurement results for task sets with task sets even near that size. This was mainly due to an effect of the Linux interrupt return path, that is responsible for actually calling the schedule function in the kernel: The hrtimer based interrupt service routines (ISR) themselves were triggered

as accurate as expected, but I experienced a delay from the (ISR) to the call of my scheduling function that was affected of a heavy time-jitter.
This jitter was strong enough, that task sets with an small absolute system slack were unable to be handled correctly by the UGLY scheduler.
Also I measured task sets where some jobs needed way more than their usual WCET. Somwhat periodic spikes were overlaid to an otherwise dense distribution of execution times as it can be seen in Figure 10.

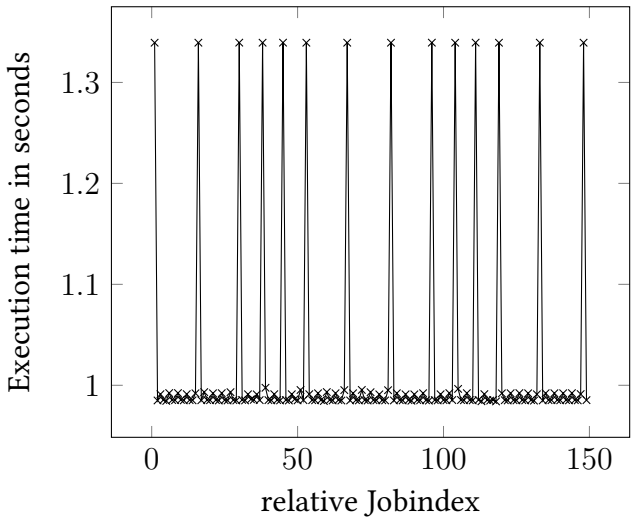


Figure 10: Periodic spikes in execution time of jobs

Most unfortunate I did not have enough time left to instrument the kernel in a way to find out about the reasons for these disabilities that occurred on the test system.

6 Conclusion

To summarize my work, I will give a short wrap-up of what has been achieved with this thesis, what unfortunately has not and what concludes from those results. As always, hindsight is easier than foresight and thus I will close with some suggestions that could improve research and use of a follow up implementation to the next level of a *Consolidate-to-Idle* scheduler.

6.1 Summary Of Results

The goal of this thesis was the initial exploration of the scheduling paradigm *Consolidate-to-Idle*. I was able to show that—at least for some task sets—*Consolidate-to-Idle* really fulfilled the hope to save energy *for free*, that is purely in software and without sacrification of real time confirmations that conventionally *Race-to-Idle* schedulers guarantee. This was shown by the restrictedly successful implementation of a partitioned-EDF based *Consolidate-to-Idle* scheduler using an adapted slack computation algorithm. On a contemporary PC platform measurements of energy consumption with the CPU provided Intel *RAPL*-counter (RAPL) were achieved in a task model synchronous manner by the implementation of simple kernel interface for RAPL and a model conforming measurement task. Experienced energy consumption was in the order of 7% lower than on the *Race-to-Idle* EDF based scheduler for some synthetic workload. But the utilization range search has not been comprehensive yet and higher values are expected.
As it turned out, the implementation in the Linux kernel led to unforeseen problems. Most unfortunate is the scheduling latency that occurs between signaling mechanism and actual scheduling action. Therefore the implementation inhered latency and jitter that prevented the model-conform execution of short-period-task sets. Caching effects were thus concealed. A model adaption that includes this delay time could improve this behaviour, but it stays unclear how small the achievable minimal period length can be.
On the positive side Linux turned out to be flexible enough to implement a task model that in its behaviour significantly differs from usual one. Also the mechanism for energy conservation through ACPI sleep-state exploitation on idle CPUs worked without modification.

6.2 Left Open Improvements and Future Research

In my opinion the next version of a *Consolidate-to-Idle* scheduler should be applied in real-time operating systems with a kernel designed more towards a deadline driven real-time task model than Linux. The problems that arose from the time delay between signaling of a scheduling concerning event and the actual execution of the task selection are not easy to overcome in my current implementation and Linux at all, but should be considered as an important concern for future re-implementations. Due to this delay time effects I have not been able to examine effects that are cache hierarchy related. A model adaption that includes this delay time could improve this behaviour, but is a complex task to achieve practically.
It would be also interesting to compare the *Consolidate-to-Idle* scheduling paradigm to the slack based approach [2] directly. Especially the result of a meta strategy that can pick between voltage scaling and consolidation dependent on the load situation on a system is promising.
Future implementations of a *Consolidate-to-Idle* scheduler like mine could also benefit from the two proposed optimisations for slack reclamation: First, the minimal extension to the slack computation for a full reclaims of slack won by faster execution than WCET planded (see Section 3.2.3). This was left out in the implementation, since it is not so important in the synthetic benchmark environment. A system could truely benefit from more slack, but is not expected to perform in a totally different way.
The second optimisation in slack gathering that was left out in the implementation was progress. One possibility was showed how progress could be addressed with the partitioning of jobs in to smaller worst case grains. In real world applications it may be for some applications easy, for some others infeasible to find such partitions that can be used as discrete points of progress.
Additionally I suggest a modular implementation of an automatic task set partitioning strategy. In Davis [4] some heuristics for this NP-hard problem are described and compared. As partitioned scheduling is a well known topic in the research of the last 40 years, I do not expect a scientific breakthrough, but simply a highly improved usability of such a test system.

Symbol Reference

f_s	Safety factor for WCET estimation
m_s	Additive overhead compensating margin for WCET estimation
I	Idle time in current hyperperiod
ST	Time used for stolen jobs
Z_i	Slack computing partition including J_{ci}
d_i	Relative deadline of job J_i.
P_i	CPU core number i
$se_{i,p}$	The p^{th} Worst-case grain, that job J_i is modeled of.
J_i	A specific job with index i.
$J_{i,j}$	The j^{th} job of the task T_i.
r_i	Spawn-time, release-time of J_i.
e_i	Time that job J_i needet to complete.
e_i^+	Worst-case execution time of J_i.
e_i^+	Worst-case execution time of J_i.
ξ_i	Runntime of job J_i accumulated.
T_i	A specific task i.
n	Number of tasks in a taskset.
H	Length of the hyperperiod of a taskset.
N	Nuber of jobs in the hyperperiod of a taskset.
$\sigma_i(t_c)$	Slack of job J_i at time t_c.
$\sigma(t_c)$	System, or runqueue slack at time t_c

Acronyms

ACPI	Advanced Configuration and Power Interface
CFS	Completely Fair Scheduler
CPU	processing unit
EDF	earliest-deadline-first
PID	process-ID
RAPL	running average power limit
WCET	worst-case execution time

Glossary

(CPU-) scheduler
A scheduler is the planning program, that decides in what CPU and how long all ready threads get coputing time on a CPU. For each CPU a scheduler always knows which task is to be run, when this decision has to be made. For this decision the scheduler creates the schedule.

Completely Fair Scheduler
This is the default scheduler in Linux since kernel version 2.6.23 and as its predecessor, the $O(1)$-scheduler initially developed by Ingo Molnár [11]

ioctrl
Ioctrls are the numbered special functions, that a device file in Linux may provide. This can be used to implement an interface for userland programs to kernelmode device drivers that export an device file. A userland programm can invoke the ioctrl file function with the specific number providet and no distinct systemcall for each driver or kernel function needs to be asigned.

kernelspace
Operating system provided priviledge mode of kernel programms. In Linux

on x86 this is all code, that is running under the hardware providet privilege separation *ring 0*. The CPU scheduler and device drivers are a typical part of kernelspace in Linux.

SCHEDULE

A plan how to distribute the available processing resources to all jobs in a given task set.

THREAD

A thread is a distinct path of execution. In the context of the Linux Kernel thread a process are sometimes used synonymous and threads are distinguishable by their unique number that is called process-ID (PID). But although their identifier is named like this, a *proces* may consist of multiple threads ,sharing the same address space.

USERLAND

Operating system provided priviledge mode of normal, unprivileged programs. In x86 and x86 computers Linux implements the restrictions in this mode by running all userland programs in the hardware providet privilegde separation *ring 3*. The term *userland* may refere to the entire collection of unprivileged software a system consists of.

References

[1] *Buildroot Linux distribution, up to git commit 3d680bf0dd0bd0f62dad91c9bf8c7657ebc5f59*, 2013.
URL: `http://buildroot.org`.
55

[2] Chen, Jian-Jia, Chuan-Yue Yang and Tei-Wei Kuo: *Slack reclamation for real-time task scheduling over dynamic voltage scaling multiprocessors.*
In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, pages 8--pp. IEEE, 2006.
22, 64

[3] *Supporting multiple CPU idle levels in kernel: cpuidle sysfs*, 2013.

URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/cpuidle/sysfs.txt.
60

[4] Davis, Robert I and Alan Burns: *A survey of hard real-time scheduling for multiprocessor systems.*
ACM Computing Surveys (CSUR), 43(4):35, 2011.
22, 28, 64

[5] Dudani, Ajay, Frank Mueller and Yifan Zhu: *Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints.*
SIGPLAN Not., 37(7):213--222, June 2002.
URL: http://doi.acm.org/10.1145/566225.513865.
22

[6] Hähnel, M., B. Döbel, M. Völp and H. Härtig: *Measuring energy consumption for short code paths using RAPL.*
Proc. GREENMETRICS'12, London, UK, 2012.
URL: http://os.inf.tu-dresden.de/papers_ps/greenmetrics2012-rapl.pdf.
49, 50

[7] Hewlett-Packard Corporation ,Intel Corporation, Microsoft Corporation,Phoenix Technologies Ltd., Toshiba Corporation: *Advanced Configuration and Power Interface Specification Rev 5.0*, 2011.
URL: http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf.
51, 59

[8] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A & 3B): System Programming Guide*, 2011.
URL: http://www.intel.com/Assets/PDF/manual/325384.pdf.
50

[9] Karam, Lina J, Ismail AlKamal, Alan Gatherer, Gene A Frantz, David V Anderson and Brian L Evans: *Trends in multicore DSP platforms.*
Signal Processing Magazine, IEEE, 26(6):38--49, 2009.
11

[10] Liu, Jane W. S. W.: *Real-Time Systems.*

Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
13, 16, 17, 27, 31, 48

[11] *LKML Discussion[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]*, 2007.
URL: `http://lwn.net/Articles/230501/`.
18, 20, 66

[12] Marr, Deborah T: *Hyper-Threading Technology Architecture and Microarchitecture.*
Intel Technology J., 6(1), 2002.
13

[13] Mezzetti, Enrico, Niklas Holsti, Antoine Colin, Guillem Bernat, Tullio Vardanega and others: *Attacking the sources of unpredictability in the instruction cache behavior.*
In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.
38

[14] Pallipadi, Venkatesh, Shaohua Li and Adam Belay: *cpuidle: Do nothing, efficiently.*
In *Linux Symposium, June*. Citeseer, 2007.
51

[15] Tia, Too Seng: *Utilizing slack time for aperiodic and sporadic requests scheduling in real-time systems.*
PhD thesis, University of Illinois at Urbana-Champaign, 1995.
30, 33, 47

[16] Van Berkel, CH: *Multi-core for mobile phones.*
In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260--1265. European Design and Automation Association, 2009.
11

[17] Williamson, David F, Robert A Parker and Juliette S Kendrick: *The Box Plot: A Simple Visual Method to Interpret Data.*
Annals of Internal Medicine, 110(11):916--921, 1989.
58