Diplomarbeit

Managing Overload with ATLAS Real-Time Scheduling

Sebastian Wagner

23. Februar 2017

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter: Dr. Michael Roitzsch
M.Sc. Hannes Weisbach

Erklärung

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

Dresden, den 23. Februar 2017

Sebastian Wagner

Abstract

This work presents an implementation of overload management for the ATLAS soft real-time scheduler. In the scope of my thesis, I examined five different strategies for distributing a required cutback among jobs of an overloaded schedule in ATLAS. Information about the current load situation is communicated with a system call from the kernel implementation to the runtime in userland.

In the userland, a load manger was implemented, that uses this information in conjunction with functional alternatives and iterative refinement functions to adapt to varying load situations. Both the kernel component, as well as the load manager, were evaluated using three different benchmarks.

Task

The ATLAS system was developed at the TU Dresden OS Chair to explore an interface, where applications explicitly submit individual real-time jobs. This policy allows for more flexible applications, but sacrifices restrictions necessary for admission testing. Therefore, a system scheduled by ATLAS can experience overload.

This thesis should augment the ATLAS infrastructure with overload management facilities. Using the knowledge of future job executions, overload should be detected and communicated to applications in a useful way. The thesis should discuss design alternatives for this communication mechanism, which may include modifications to the ATLAS kernel scheduler, additional user-level components, or changes to the ATLAS runtime library. One communication method should be selected and implemented.

To mitigate the overload, execution time allocations of applications must be throttled or otherwise penalized. Different cutback policies like absolute, proportional, or utility-based fairness can be employed. At least two different policies should be implemented and compared using benchmark applications. The evaluation should analyze the impact on execution time reservations and violations of application timing requirements.

Contents

1 Motivation

Current operating systems (OS) like Windows, OS X, and Linux possess the ability to concurrently execute multiple applications. This enables users to have services running in the background and also start multiple applications in the foreground that they want to use. For most machines and use cases, the amount of applications executing in parallel is way higher than the amount of tasks the hardware is actually able to concurrently execute. Hence, temporal multiplexing for the execution of tasks on the central processing unit (CPU) has to take place. This is done via scheduling in the OS. [TB14]

Applications may have different requirements with respect to how often they start work on the CPU, how much CPU time this work uses, and until what point in time the work has to be finished. For example, a video player has to load the data for a frame, decode it, and display it at a certain point in time. If the video player does not achieve to display the frame in time, the user may recognize a degradation of the Quality of Service (QoS) of the video stream.

In current OS, the scheduler usually does not know about the timing requirements of applications. At best, it is possible to indicate some form of priority for the task so the scheduler can better decide which task to run next. However, although priorities can express which task to favor when selecting the next task to execute, they cannot represent deadlines.

Classic real-time OS research introduced hard real-time systems as a solution to this problem. For applications with varying load hard real-time systems overestimate resource usage, as the worst case execution time (WCET) has to be assumed. Soft real-time systems are a model more suited for applications like video players. While scheduling tasks with specific deadlines and execution times, in soft real-time systems those deadlines do not always have to be met. The Auto-Training Look-Ahead Scheduler (ATLAS) is such a soft real-time system. ATLAS relieves programmers of the need to supply the execution times for their applications. Instead, ATLAS predicts them using a linear auto-regressive predictor.

In ATLAS, jobs arrive while the scheduler is running and are unknown beforehand. So, ATLAS can be classified as a clairvoyant on-line scheduler, in which it is not possible to create a schedule a priori. The schedule ATLAS executes depends on the quality of the execution time predictions. Thus, the schedule is not fixed and situations may occur in which the schedule is overloaded.

Overload might develop either voluntarily or involuntarily; the former, due to the user being unaware of the current system load and submitting too many jobs, the latter, for example, if a malicious process tries to claim a bigger share of the available CPU time by submitting either too many jobs or jobs with largely exaggerated execution times.

If such a situation occurs, usually the overall QoS of the system declines due to jobs missing their deadlines. Overload can be counteracted by applications that are aware of the load situation in the system and react to it. Some possibilities to react to high load situations that will be highlighted in this thesis include specifying functional alternatives and functions of iterative refinement.

Functional alternatives provide, for certain load intensive functions, other functions that calculate less precise results. Albeit the result being slightly inaccurate, the application can still work with it. Iterative refinement functions are designed as loops, that yield more precise results the longer they run. In these loops, each iteration uses the result of the previous iteration to improve the precision. The suitability of both of these concepts of imprecise calculation for overload mitigation in ATLAS will be examined in this thesis.

2 Scheduling

This chapter will present the basics for the rest of this thesis. First, a short overview of how scheduling works in hardware is given in Section 2.1. After that, Section 2.2 introduces the concept of real-time applications. Sections 2.3 and 2.4 show how scheduling in Linux and ATLAS is implemented. Finally, Section 2.5 will give examples on how to practically use the knowledge acquired earlier.

2.1 A Primer on Scheduling[1]

Applications consist of a finite number of instructions to execute. Those instructions can read, write, and modify memory. To prevent applications from accessing other applications' memory, OS use virtual memory. Physical memory is no longer addressed directly but by using virtual addresses that map to physical memory. The translation of virtual addresses to physical addresses is done by the memory management unit (MMU). Virtual memory offers two advantages: first, memory-modifying instructions of one application cannot influence other applications' memory; second, applications can be programmed as if they owned the whole address space.

When switching from one application to another, the memory mapping for the task's address space and register values have to be exchanged. This is called a context switch and performed by the scheduler. To regain control of the CPU in the first place, the scheduler uses a programmable interrupt timer (PIT)[2] that emits an interrupt after a certain amount of time. The interrupt is then routed via the programmable interrupt controller (PIC) to the CPU[3], the interrupt handler of the OS processes it, and control is passed to the scheduler. The scheduler selects the next task to be executed and restores this task's state. It then passes control to the task, which continues execution, and after a certain time, the same procedure is repeated, thus creating the impression that multiple applications are executed in parallel. This mechanism is usually referred to as multitasking. On systems that consist of more than one core, of course, applications can actually run in parallel. But even in this case, multitasking is necessary if more applications than cores should run in parallel.

Another abstraction mechanism used by OS is threading. Threads allow applications to define multiple control flows that are executed in parallel or pseudo-parallel. In contrast

[1] This section is intended for readers unfamiliar with the field of process scheduling in operating systems. Thus explanations are simplified.

[2] An integrated circuit (IC) located on the mainboard.

[3] Modern systems use an advanced PIC (APIC) that has a split design with the local APIC (LAPIC) part integrated directly into the CPU. The LAPIC part also contains a high-resolution timer.

to tasks, threads of the same task operate on the same address space and therefore have access to the same data.

Due to the high cost of context switches, OS schedulers try to avoid them. Response times (e.g. the time until an application can react to an event like a key-press) on the other hand should also be kept low by the scheduler. This presents a dilemma. If the scheduler switches too often, overhead induced by context switches increases. If it switches too rarely, response times render the system unusable.

Sometimes, applications have to wait for I/O operations to finish and thus cannot continue computation. Also, there are applications that are designed to carry out actions periodically. In those cases the application can voluntarily yield control to the scheduler which in turn can schedule another task that actually uses the CPU. This mechanism, called a voluntary context switch, is orthogonal to the aforementioned involuntary context switch caused by the timer interrupt. When control is passed to the scheduler by an involuntary context switch, an application is said to be preempted.

2.2 Real-Time

Applications such as video players, have constraints on when results of their computations need to be delivered.[4] Those applications—usually referred to as real-time applications—encapsulate work that has to be completed into so called *jobs*. Jobs are characterized by their respective deadlines and execution times. While the correctness of non-real-time applications only depends on the delivered result, a real-time application's correctness also depends on *when* this result is available.

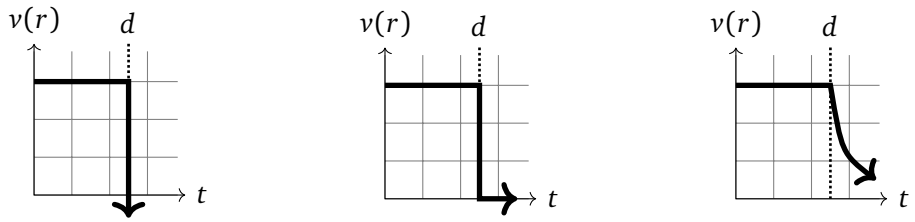
2.2.1 Soft, Firm, and Hard Real-Time

Jobs of real-time applications can be differentiated by how computations that miss deadlines affect the system. The value of a result of the computation differs depending on *when* it arrives. This behavior is depicted in Figure 2.1 with the x-axis representing the time t the result arrives and the y-axis the value of the result of the function $v()$ to the system as a whole [But11]. How the value of the result changes after the deadline d is determined by the type of the real-time task at hand.

Hard Real-Time Jobs

Hard real-time jobs may never miss a single deadline. If one is missed, the whole system is useless and there is no point in continuing any computation using this system. One prominent example would be an air bag that, if it failed to trigger *in time*, would be

[4] Staying in the example, video players should have each frame decoded at the point in time it is supposed to be displayed.



(a) hard (b) firm (c) soft

Figure 2.1: Different types of real-time jobs

pointless. As seen in Figure 2.1(a), the result loses all value and the system ceases to function, if a single deadline is missed[5].

Firm Real-Time Jobs

Firm real-time jobs can tolerate misses of deadlines on some jobs. If a job misses its deadline, the computation done by that job is useless, but the system can continue to function, if it does not miss too many deadlines in a specific period of time. For the overall QoS of the system, it is essential that deadline misses do not occur too frequently. For example, a video player could miss deadlines on some frames—and compensating for this by reusing the previous frame—without the user noticing this. If there are too many deadline-misses, the QoS would suffer and users may perceive the degradation of quality. The Plot 2.1(b), depicts this as $v(r)$ having no value after the deadline and continuing to do so.

Soft Real-Time Jobs

Soft real-time jobs may, just like firm ones, miss their deadlines. In contrast to firm real-time jobs, results of the computation done by the job are still useful after the job has passed its deadline, though their value decreases gradually. Figure 2.1(c) displays this by letting $v(r)$ approach zero after the deadline d.[6]

2.2.2 Aperiodic, Periodic, and Sporadic Tasks

Tasks can further be classified by how often and how regularly they start jobs. Classic real-time research groups jobs into the classes *aperiodic*, *periodic*, and *sporadic*.

[5]Though not mathematically correct, it is a nice representation for a failing system.

[6]The partial function used after the deadline is application-specific and may be different from what is depicted in Figure 2.1(c).

Aperiodic Tasks

Aperiodic tasks produce work only once. So, they (or their job) are characterized by their WCET and relative deadline. As they are not repeated, in contrast to periodic or sporadic tasks, they lack a period or minimum inter-arrival time.

Periodic Tasks

Periodic tasks regularly emit jobs with a certain WCET. How often a task starts a new job is denoted by its period. Additional to the WCET of these jobs, they have a relative deadline. Their scheduling is extensively researched and there are many implementations, e.g. by classic scheduling algorithms like earliest deadline first (EDF) or rate monotonic scheduling (RMS)[7].

Sporadic Tasks

Sporadic tasks, like periodic tasks, emit more than one job. While periodic tasks emit jobs in a regular fashion, the arrival times of jobs of a sporadic task are only bounded by a minimum inter-arrival time. In the worst case, the scheduler has to plan for jobs arriving at exactly the period defined by said minimum inter-arrival time. This has the drawback that it usually largely over-estimates actual resource usage.

2.3 Linux

Linux is a unixoid operating system kernel initially developed by Linus Torvalds. After Torvalds released its source code in 1991, it has grown to include more than 19 million lines of code[8] distributed under the GNU General Public License (GPL). As a large portion of this code base consists of drivers, Linux supports a wide variety of hardware. Linux runs on many instruction set architectures (ISAs) from rarely used ones like PArisc to widespread ones like x86 and ARM, thus making it applicable for many usage scenarios. [Lee15]

Development of the kernel is done in a distributed way using the git version control system with contributing developers from all over the world. Due to its open source nature, it is possible to extend and modify the Linux kernel. This was, for example, done by Google with Android, which is now the most commonly used OS on mobile phones. Linux is also widely used on servers. An overview of the OS used to run web servers shows that 66.6 % of them use Unix-like systems. Of these 66.6 % of systems, 55.2 %[9] use Linux as their OS. [W3T16]

[7] Formally, both describe different methods for assigning priorities. The scheduling is then done based on these priorities.

[8] As at Linux 4.0.

[9] As at 11/2016. Note, that this study also shows 43.7 % systems for which the study was not able to determine the OS. The actual share of Linux may be higher.

2.3.1 Processes

In the Linux kernel, both processes and threads are represented by tasks. Implementation-wise, the difference is that threads share their parents' address space while newly created processes are assigned a new memory mapping. Internally, tasks are represented by a structure `task_struct` that contains the following data:

- General information about the task at hand (process ID, thread group ID, state, stack pointer, exit state, memory mappings, whether the task is being traced).
- Information about how the task should be scheduled (CPU affinity, scheduling class, the run queue the task is on, priority).
- Signal processing information, like signal handlers and the mask of blocked signals.
- Statistics and accounting information (number of context switches, time spent in system and user mode).

2.3.2 Schedulers

Linux includes multiple schedulers that serve different purposes. For implementing those schedulers, the Linux kernel provides a structure `sched_class` that each scheduler implementation has to define[10]. Next, I will first describe how scheduling in Linux works in general. After that, run queues and the scheduler interface structure are described. Finally, I will give an overview of the different schedulers implemented in Linux.

Scheduler Invocation

The scheduler of the Linux kernel is invoked by calling `schedule()`. When invoked, the scheduler selects the next task to run and executes a context switch to the newly selected task. In case it selects the just preempted task, the scheduler passes control back to this task, saving the unnecessary context switch. The `schedule()` function itself is invoked when

- a task switches into the sleep state or
- wakes up from the sleep state,
- and regularly via a timer interrupt.

Run Queues

A run queue is a container that holds tasks known to the scheduler. There exists one structure `rq` for each CPU. Each `rq` holds scheduler-specific run queues for each sched-

[10]Though some fields may be set to `NULL`.

uler implementation. Scheduler-specific run queues are usually named <scheduler name>_rq and do not necessarily have to be queues; the run queue for the Completely Fair Scheduler (CFS), cfs_rq, for example, is implemented as a tree.

Interface

The schedulers included in Linux are described by the structure sched_class that contains a pointer to the next sched_class. This is used to order scheduling classes from higher to lower sched_classes[11]. Tasks of a higher sched_class get to run first, while tasks of a lower sched_class only get to run when none of the tasks of higher classes are runnable. The rest of the structure consists of several function pointers to the corresponding implementation[12] [See13].

- enqueue_task and dequeue_task are responsible for inserting a task into and removing one from the run queue of the current sched_class.
- yield_task is called when a task voluntarily yields the CPU. For example, this function can be used, when a thread has to wait for another thread or a resource.
- yield_to_task enables a calling task to yield the CPU in favor of another task in the same task group. This is currently only used in CFS. [van11]
- check_preempt_curr takes a task as its argument and checks whether the given task can preempt the current task. If both tasks are of the same class, check_preempt_curr of the current scheduler implementation is called. Otherwise it checks, if the given task has a higher sched_class than the current task.
- pick_next_task: This function is used to find the next task to run. Depending on the actual sched_class implementation, pick_next_task may use a heuristic to determine which task to run next or just pick the first task in the sorted backing data structure of the scheduler.
- For a descheduled task and (possibly) its parent task, put_prev_task updates the accounting information.
- When a task changes its class, set_curr_task is called to set the current task of the run queue of the scheduler and to update accounting information of this task.
- task_tick is invoked regularly by the scheduler timer. It updates the statistics about the running task and its parents like the amount of CPU time the task has received.
- Whenever a task creates a child task (a child-process or a thread from the user-land's point of view), task_fork is called from the creating task's context. This function receives the new task as an argument and is responsible for enqueuing it into the run queue of the CPU it is intended to run on.

[11] If a sched_class *A* points to sched_class *B* via this pointer, *A* is a higher scheduling class than *B*.
[12] Linux kernel source code, Version 4.0, kernel/sched/core.c

- `switched_from` and `switched_to` are called, when the task at hand is moved from one `sched_class` to another one. `switched_from` is called for the previous class and `switched_to` for the new one.
- `prio_changed` is invoked when the priority of a task has changed. This might preempt the current task.
- For updating runtime statistics of the current task, like received execution time or when execution of the task has started, `update_curr` is used.

If the kernel supports symmetric multiprocessing (SMP), the scheduling class can define implementations for functions related to it. For example, these include `select_task_rq` which selects the run queue and thus CPU a waking task is supposed to run on. By changing to an appropriate run queue, load balancing can be implemented in Linux. Another possible function pointer to define is `migrate_task_rq`, which is called immediately before migrating a task to a new run queue.

By using the `next` field of the `sched_class`, the individual scheduler implementations form a hierarchy, which is depicted in Figure 2.2. Only if no tasks of higher `sched_classes` are runnable, tasks of lower ones are considered for selection. The current[13] hierarchy from higher priority schedulers to lower ones is as follows[14] [Ces14, Tor15]:

- `stop_sched_class`: The *stop* scheduling class is a special class. It can only schedule one stop task per run queue. This task, as it is in the highest scheduling class, preempts every other task and can be preempted by no task. The stop task is used when migrating tasks between CPUs.
- `dl_sched_class`: The *deadline* scheduling class implements Abeni and Buttazzo's concept of the Constant Bandwidth Server (CBS) [AB98].
- `rt_sched_class`: The *realtime* scheduling class implements the `SCHED_RR` and `SCHED_FIFO` scheduling classes described in the Portable Operating System Interface (POSIX) specification.
- `fair_sched_class`: This class represents *CFS*. It can be used with the different scheduling policies *batch*, *normal*, and *idle*.
- `idle_sched_class`: The *idle* scheduling class is like *stop* a special scheduling class that can schedule one idle task per CPU. This task can be used to detect, when the CPU can be put into power saving states.

Schedulers implemented in Linux

$\mathcal{O}(1)$-Scheduler Version 2.6 of the Linux kernel introduced a scheduler that had a runtime complexity of $\mathcal{O}(1)$ with regard to the number of scheduled tasks. It was implemented using two lists, one holding tasks that are about to run and the other holding tasks that have already run. The scheduler would always take the first process, execute

[13] As at Linux 4.0.
[14] See the next section about more detail on the deadline, realtime, and fair scheduling classes.

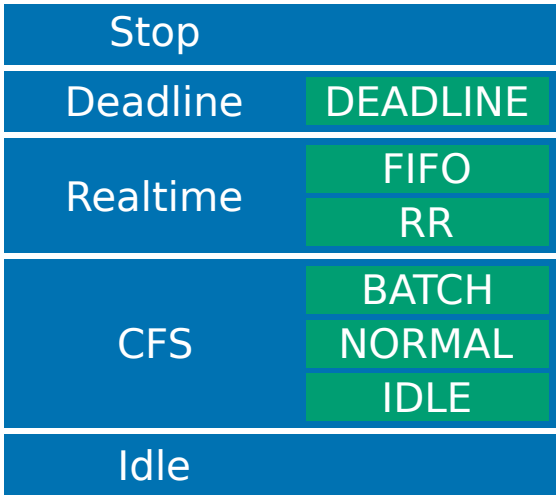


Figure 2.2: Scheduling classes implemented in Linux [Wei16]

it, and insert it into the second list. When all the processes had run, the scheduler would swap those two lists and repeat the process.

CFS Starting with version 2.6.23, Linux replaced the $\mathcal{O}(1)$-scheduler with the Completely Fair Scheduler (CFS). It uses a red-black tree as the data structure for its run queues. Red-black trees offer the advantage that insertion, deletion, and searching in the tree can be done in $\mathcal{O}(\log n)$[15] in the worst case.

A task waiting for I/O-operations (like reading from a file descriptor) and thus blocking most of the time is said to be I/O-bound. Contrary, tasks that heavily utilize the CPU are called CPU-bound.

CFS offers a notion of sleeper fairness, meaning that I/O-bound tasks and ones that often voluntarily sleep are prioritized when selecting the next task to run. This decreases the time an I/O-bound task has to wait until it gets selected, with the intention of reducing response times for interactive applications.

For scheduling different types of applications, CFS offers three different scheduling policies [BC05]:

- `SCHED_NORMAL` is used for regular applications that do not need to be scheduled in a special way. It offers the aforementioned sleeper fairness and is thus suited for interactive as well as non-interactive applications.
- `SCHED_BATCH` reduces the amount of times a task is preempted. Therefore, this policy allows better use of caches while sacrificing interactivity.
- `SCHED_IDLE` can be used for non-interactive background tasks, which are only run when no task of the other scheduling policies is runnable.

[15] For $\mathcal{O}(\log n)$, n denotes the number of elements in the tree.

Realtime With the realtime class, Linux implements the `SCHED_FIFO` and `SCHED_RR` policies specified in the POSIX specification. [IEE08]

`SCHED_FIFO` implements a scheduler that executes tasks in first-in-first-out-order. Tasks do not get preempted unless they voluntarily yield the CPU or block.

`SCHED_RR` uses time slices of a length specified by the sysctl-variable `sched_rr_timeslice_ms`. After each task's timeslice is used up, it switches to the next task of the same priority in a round-robin (RR) fashion. Only when no tasks of higher priorities are runnable, tasks of lower priorities are considered for execution.

Deadline The deadline scheduling class is an implementation of the CBS concept, described by Abeni and Buttazzo [AB98]. It offers *temporal isolation* between its tasks and against tasks of lower scheduling classes. To avoid starvation of the system by tasks of this class, deadline tasks use at most 95 % of the available CPU bandwidth. The remaining 5 % are used for lower scheduling classes and the Linux kernel itself.

2.4 ATLAS

The Auto-Training Look-Ahead Scheduler (ATLAS) was developed at TU Dresden's operating systems chair. ATLAS extends the Linux scheduler infrastructure by a soft real-time scheduler for aperiodic jobs. Usually, an aperiodic real-time scheduler has to be informed about the execution times and deadlines of its jobs. This might pose a problem, if the programmer only knows when the desired result needs to be available. To find the execution time of a job, WCET analysis has to be performed. Even for simple jobs, WCET analysis might not always be feasible[16] [SA00, WEE+08]. Also, even if a WCET can be found, it is dependent on the used hardware and thus highly unportable.

ATLAS takes another approach. It handles soft real-time jobs, so execution times do not always need to be precise. This enables ATLAS to predict execution times for its jobs, thus relieving the programmer of the need to do so. The prediction of execution times is implemented in ATLAS' runtime `atlas-rt`[17].

ATLAS resides between the DL scheduling class and CFS in the Linux scheduler hierarchy. Figure 2.3 shows the three different scheduling policies that ATLAS uses, which will be described later on: *ATLAS*, *EDF Recovery*, and *CFS Recovery*.

The runtime provides a partial implementation of Apple's Grand Central Dispatch (GCD) queues. GCD exposes an interface that allows the programmer to express concurrency in an application. If the used hardware supports concurrent execution, GCD makes use of it in a sensible way. This includes issues like thread creation and placement of threads onto specific CPUs. With ATLAS, the use of GCD-like queues hides the prediction of execution times from the programmer.

[16] Finding the WCET of an arbitrary application is equivalent to the halting problem.

[17] Both the source code of ATLAS and atlas-rt can be found on Hannes Weisbach's GitHub page: `https://github.com/hannesweisbach`. The repositories are named `linux-atlas` and `atlas-rt` respectively.

Figure 2.3: ATLAS in Linux' scheduler hierarchy [Wei16]

ATLAS runtime provides wrappers for ATLAS' system calls, like the libc provides wrappers for most[18] system calls of the vanilla Linux kernel. This allows the programmer to issue the system call without having to utilize `syscall()` with the corresponding system call number.

2.4.1 Kernel

The kernel component of the ATLAS framework offers several system calls that can be used to inform the scheduler about new jobs, update already submitted ones, or cancel them. The system calls are described after a short overview of how scheduling in ATLAS works.

Scheduling

ATLAS schedules using a modified latest release time (LRT) scheduling algorithm. LRT schedules jobs in a way that they are all started at the latest point in time at which they can still meet their respective deadlines. This can be realized by treating the deadlines of jobs as their release times and building a EDF schedule in reverse order. LRT, as well as EDF, is optimal in finding a feasible schedule in a non-overloaded unicore system.

By using LRT, ATLAS also schedules jobs at the latest point in time at which all jobs' deadlines can be met. ATLAS' implementation thus builds its schedule beginning at the

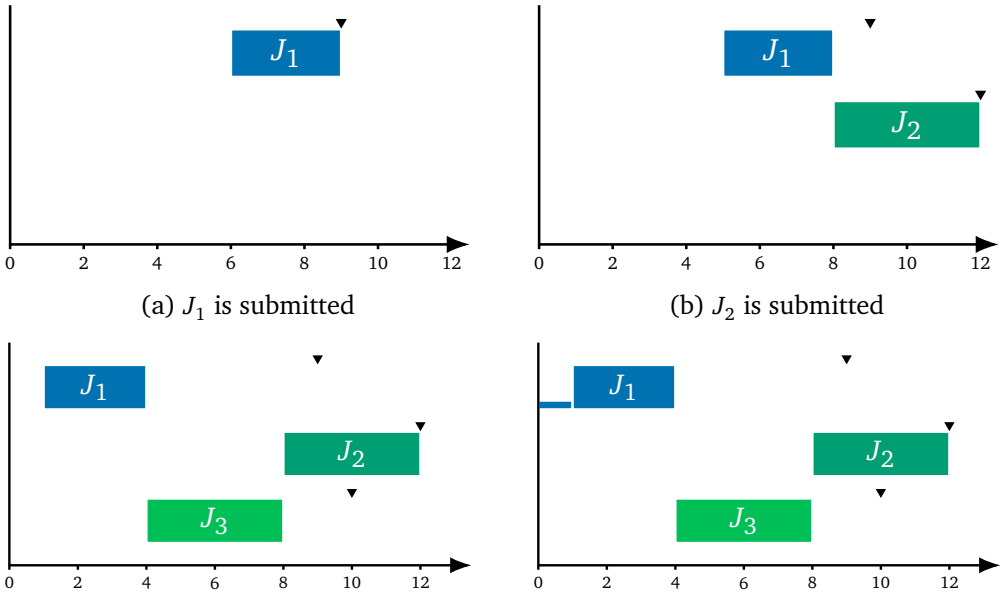
[18] For example, in glibc there is no system call wrapper for the `gettid()` system call.

latest deadline to the current point in time. ATLAS first selects the job with the latest deadline and schedules it to start at the latest time it can meet its deadline. Then the job with the next to latest deadline is placed so that it can still meet its deadline, and so on. If a new job arrives, it might be possible that jobs which have an earlier deadline have to be moved to an earlier starting time.

When a job already received all of its reserved execution time, ATLAS has to decide how to handle this job. If the job has not finished but also did not block, it gets demoted to `CFS`.

If jobs block and thus miss their deadlines, ATLAS tries to help them to catch up with their schedule. Jobs that have missed their deadlines due to blocking are scheduled in EDF order in the `EDF Recovery` band of ATLAS. They stay there for at most the time they have spent blocking. If a job has not finished when its time in `EDF Recovery` is used up, it is demoted to `CFS Recovery` in the `CFS` scheduling policy of ATLAS. Also, if a job missed its deadline not due to blocking but due to taking too long, it is demoted to `CFS`.

In case the schedule is not overloaded[19] and no job needs to recover from blocking, ATLAS allows jobs to *pre-roll*. This means that the first job in ATLAS' schedule is executed in CFS, if it does not need to be started in ATLAS yet.



(a) J_1 is submitted

(b) J_2 is submitted

(c) J_3 is submitted

(d) J_1 is allowed to pre-roll

Figure 2.4: Example of scheduling in ATLAS

This is exemplified in Figures 2.4(a) to 2.4(d). The current time in the example is 0:

- Figure 2.4(a): Job J_1 arrives with its deadline[20] at time 9 and a predicted execution time[21] of 3 time units. The job is initially scheduled to start at time 6 and end

[19] An overloaded schedule in ATLAS is one in which the first job has negative slack.
[20] Deadlines are denoted by the triangles.
[21] Execution times are denoted by the width of the rectangle.

at time 9, just in time to meet its deadline.

- Figure 2.4(b): Another job, J_2, arrives. As it has its deadline at 12 and an execution time of 4, it gets scheduled to start at time 8. As the new job would run into J_1's reservation, J_1 has to be rescheduled to start at an earlier point in time, namely 5.
- Figure 2.4(c): A job, J_3, with a deadline between the two scheduled jobs, ATLAS already knows about, arrives. As ATLAS builds its schedule starting with the job that has the latest deadline, first J_2 is scheduled at the latest time possible. Then, the new job, J_3, is added to start at time 4, immediately preceding J_2. Again, job J_1 has to be moved to an earlier starting point, at time 1.
- Figure 2.4(d): As no further jobs arrive and the first job J_1 does not yet have to start, ATLAS allows it to pre-roll. J_1 is allowed to execute in CFS, along with regular CFS tasks. If it receives CPU time while pre-rolling, this might lead to J_1 finishing earlier[22]. The next job in the schedule, J_3, would have slack and might also be allowed to pre-roll. Depending on how much CPU time jobs that pre-roll receive in CFS, the slack and thus the possibility to pre-roll can propagate to later jobs.

To summarize, the scheduling policies used in ATLAS are as follows:

- `ATLAS` is used for regular jobs between their scheduled starting time and scheduled deadline.
- `EDF Recovery` schedules jobs that have overrun their deadlines due to blocking of the job.
- `CFS` handles `CFS Recovery` and pre-rolling jobs.

System Calls

To interact with the scheduler part of ATLAS, several system calls are implemented that are described in the following:

- `atlas_submit(pid, id, exectime, deadline)`
  This system call informs ATLAS about a new job with ID `id`. `id` can be arbitrarily chosen, while `pid` is supposed to be the *thread ID* of the thread that will execute said job[23]. The `exectime` argument is the duration, which a job is expected to run, while `deadline` is the absolute deadline of the job.
- `atlas_next(next_id)`
  The ID of the next job to execute is written to where the parameter `next_id` points to. In case there are no jobs scheduled to be executed next, this call blocked in earlier versions of ATLAS. More recent versions do not block anymore, but always

[22] This assumes the submitted execution time was not too far off.
[23] That is the thread whose execution will be accounted to this job.

return the number[24] of written job IDs or a negative error code, if the system call failed.

- `atlas_update(pid, id, exectime, deadline)`
  To update either the execution time, the deadline, or both, the `atlas_update` system call can be used. If either `exectime` or `deadline` are not `NULL`, the value for this job will be updated. This might be used by applications to adapt to overload situations by reducing the amount of work they have submitted.

- `atlas_remove(pid, id)`
  The `atlas_remove` system call removes a job from ATLAS that was previously submitted to the system. This might be used by processes to cancel optional jobs in high load situations.

Together, `atlas_update` and `atlas_remove` allow applications to react to high load situations by updating or cancelling jobs they have already submitted.

For ATLAS to be aware of the thread pools used by concurrent queues, it also implements system calls to manage thread pools.

- `atlas_tp_create(id)`
  This creates a new thread pool with the given id, but doesn't yet create any threads as workers.

- `atlas_tp_destroy(id)`
  Thread pools are destroyed using the `atlas_tp_destroy` system call. All worker threads of the targeted thread pool need to have exited beforehand.

- `atlas_tp_join(id)`
  To add a worker thread to a thread pool, the `atlas_tp_join` system call is executed by the worker thread to join the pool with ID `id`. Worker threads need to be pinned to one CPU and thus may not migrate. There is no system call to remove a worker thread from a thread pool. Instead, worker threads are removed when they exit.

- `atlas_tp_submit(tpid, id, exectime, deadline)`
  By using `atlas_tp_submit`, a job is enqueued in the thread pool with ID `tpid`. Like with `atlas_submit`, the execution time and deadline have to be passed as arguments.

2.4.2 Userland

In addition to the kernel part, ATLAS includes a runtime—`atlas-rt`—that provides a convenient application programming interface (API) to interact with the kernel component. The runtime combines a predictor with an implementation of GCD-like queues to automatically predict execution times for jobs queued in it and submit them to ATLAS. The next sections will describe this API and the predictor.

[24] Currently, this can be 0 or 1.

Grand Central Dispatch

Apple's Grand Central Dispatch (GCD) queues are a mechanism to use task level parallelism. In GCD, programmers encapsulate work into so called work items that are enqueued in a dispatch queue and then started in first-in-first-out (FIFO) order. Work items may be BlocksRuntime blocks or function pointers[25]. The way work items are executed is determined by the type of dispatch queue at hand. Serial dispatch queues sequentially execute jobs that are in the queue and only start the next job, after the current one has finished; concurrent queues may start them in parallel. [App16]

Dispatch queues are implemented using the thread pool design pattern. This means, that the GCD implementation creates an amount of worker threads that is appropriate to the system at hand. For example, on a system containing four physical cores that support hyperthreading, GCD might create eight worker threads. In contrast to creating a new thread for every work item, this has the advantage of being less resource-heavy[26].

ATLAS partially implements the GCD interface by providing the class `atlas::dispatch_queue(label)`, that creates a named serial queue, as well as `atlas::dispatch_queue(label,cpuset)` for creating a named concurrent queue whose jobs are allowed to run on the CPUs in `cpuset`. Instances of these classes can be used to dispatch work items using the following methods:

- `dispatch_async_atlas(deadline, metrics, metrics_cnt, fn, args...)` The `metrics` array is used for the prediction of the execution time for this job as described in the next section. The predicted execution time and the `deadline` are then used to submit the function `fn` as a job with arguments `args` to an appropriate thread pool. The result[27] of the computation is encapsulated into a `std::future` that will yield the result when it is available.

- `dispatch_sync_atlas(deadline, metrics, metrics_cnt, fn, args...)` This method works like the previous, except it blocks and returns the result directly.

Furthermore, dispatch queues in ATLAS also provide methods to enqueue work items as non-ATLAS jobs:

- `dispatch_async(fn, args...)`

- `dispatch_sync(fn, args...)`

These methods behave like their `*_atlas` counterparts with respect to blocking. Due to ATLAS' scheduling, jobs submitted with the `*_atlas` methods are subject to reordering according to their deadline, while non-ATLAS jobs are always processed in submission order.

[25] The ATLAS runtime extends this by also allowing C++ lambdas.

[26] According to Apple, enqueuing a GCD work item only requires 15 instructions while creation of a thread from userspace takes significantly longer. [App09]

[27] Currently, the only supported return type is `void`.

Prediction

One of the advantages of ATLAS is the prediction of execution times. As `atlas-rt` is capable of predicting future execution times, the programmer is relieved of this burden while still having the advantage of using a soft real-time scheduler.

The predictor uses workload metrics that are application specific. It is the programmer's responsibility to provide a vector $m = (\ m_0\ m_1\ \cdots\ m_l\ )^T$ of sensible metrics via the `dispatch_*_atlas` runtime calls for each job that linearly correlate to the job's execution time. If the programmer fails to do so, the execution time predictions are meaningless. Internally, a history of vectors of metrics is stored in a matrix M.[28] [Roi13]

To actually predict future execution times, ATLAS uses a linear auto-regressive approach. Additionally to the matrix M, the predictor stores a vector of measured execution times t. If metrics would always directly correlate to the execution times, one could find a vector of coefficients $x = (\ x_1\ x_2\ \cdots\ x_l\ )^T$ such that

$$Mx = t$$

holds. But as they usually are not, x has to be approximated:

$$Mx \approx t$$

Which may also be written as

$$\begin{pmatrix} m_{0,0} & m_{0,1} & \dots & m_{0,l} \\ m_{1,0} & m_{1,1} & \dots & m_{1,l} \\ \vdots & \cdots & \ddots & \dots \\ m_{k,0} & m_{k,1} & \dots & m_{k,l} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_l \end{pmatrix} \approx \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_k \end{pmatrix}$$

For each row of the matrix and thus each job, the error of the approximation can be calculated as

$$r_i = x_0 \cdot m_{i,0} + x_1 \cdot m_{i,1} + \ldots + x_l \cdot m_{i,l} - t_i$$

The goal here is to minimize the error over all jobs[29]. The residual error of the overall approximation is defined[30] as

$$\sqrt{\sum_{i=0}^{k} r_i^2} = ||Mx - t||_2$$

and should be minimized. This can be approximated by formulating it as a linear least squares problem:

$$||Mx - t||_2 \rightarrow \min_x$$

[28] Here, matrices are denoted by capital letters and vectors by lower case letters.
[29] The jobs are represented by rows of the matrix.
[30] $||Mx - t||_2$ denotes the Euclidean Norm of $Mx - t$.

The solution[31] to the linear least squares problem, i.e. the vector x, can then be used with the newly provided metrics m_{k+1} to get an execution time prediction $\hat{t}$ used to submit the new job to ATLAS:

$$m_{k+1}x = \hat{t}$$

As each call to `dispatch_*_atlas` would extend the matrix by one row, computation of the coefficients x with the just presented model would be unbounded. Consequently, to avoid each subsequent call to the dispatch-function taking longer than the previous one, an updating solution is chosen[32].

2.5 Usage

To illustrate the usage of ATLAS, the next sections will give two examples of how to employ ATLAS from a programmer's perspective. Section 2.5.1 shows how to submit a job to ATLAS using plain C. Another example is given in Section 2.5.2, that shows usage of a GCD-like concurrent queue that also predicts execution times using a given metric.

2.5.1 ATLAS without its Runtime

Listing 2.1 shows how to use the system calls of ATLAS without help from the ATLAS runtime[33]. The job we want to submit is an aperiodic soft real-time job (cf. Section 2.2) with an execution time of 1 s which has its absolute deadline at 2 s. The work itself will be done in the function `do_work()`.

As we will not start a new thread (cf. Section 2.3) to execute the `do_work()` function in this example, first we get the thread ID of the current thread[34]. This thread ID will later on be submitted to ATLAS as the thread executing our job.

Also, we need an arbitrary job ID[35] and two `timeval` structures, `ex` and `dl`. The execution time is represented by `ex`, which we chose to set to 1 s, if our fictional work would execute for approximately one second. The deadline—given as an absolute deadline—is represented by `dl` and set to be 2 s from the beginning of the clock of the system.

As all prerequisites for the `atlas_submit()` system call (cf. 2.4.1) are available, we can now submit the new job to ATLAS. By calling `atlas_submit` with the arguments thread ID `tid`, job ID `jid`, execution time `ex`, and deadline `dl`, a new job with those parameters is created in ATLAS.

[31] The linear least squares problem can be solved by performing a *QR decomposition*.
[32] The matrix R of the QR decomposition is augmented by the new metrics, brought back into upper triangular form, and then the last row is dropped.
[33] Using the ATLAS kernel component without the ATLAS runtime is not supported. Nonetheless, I will show it here, as the ATLAS runtime was not used in this thesis.
[34] That is the process ID of the process executing the `main()` function.
[35] The `jid` was chosen to be `42` here.

```c
#include <stdint.h>
#include <sys/time.h>
#include <sys/types.h>

#include "atlas/atlas.h"

void do_work();

int main() {
  pid_t    tid = gettid();
  uint64_t jid = 42;
  uint64_t next;

  struct timeval ex = { 1, 0 };
  struct timeval dl = { 2, 0 };

  atlas_submit(tid, jid, &ex, &dl);
  atlas_next(&next);

  do_work();

  return 0;
}
```

Listing 2.1: Submitting an ATLAS-job from C

Next, a thread that wants to execute ATLAS jobs has to get the job ID of the job to execute from ATLAS by means of the `atlas_next()` system call[36]. This procedure is intended for worker threads that may execute multiple ATLAS jobs consecutively. As jobs may be reordered by ATLAS according to their deadline, it is necessary for worker threads to gain information on which job to execute. As no other job was submitted, `next` receives the value 42 and execution of the `do_work()` function can begin.

Depending on what time t the clock of the system currently shows, the job is scheduled differently:

- $t < 1\,\mathrm{s}$: If the job is the only job in the system and still has time until it needs to start, it is allowed to pre-roll. It gets scheduled in CFS and time received there is not accounted to its received execution time.
- $1\,\mathrm{s} \leq t \leq 2\,\mathrm{s}$: In this interval, the job is scheduled to execute as a regular ATLAS job. If it has not yet finished, it is scheduled in the `ATLAS` band of ATLAS and preempts any jobs of lower scheduling classes. Time received in this scheduling band is accounted to the job.
- $2\,\mathrm{s} < t$: The current time is later than the deadline, so the job has overrun its deadline. If the job already received the execution time it reserved, this is the job's fault and ATLAS demotes it to CFS until completion. If the job has not received its reserved execution time, it must have blocked. In this case, ATLAS helps the job

[36] Usually, the value in `next`, as well as the return values `atlas_next()` and `atlas_submit()` should be checked. For the sake of the brevity of this example, this is omitted here.

to catch up by scheduling it in the `EDF Recovery` band [37], until it has received the scheduled execution time. If the job has not finished until then, it also gets demoted to CFS.

2.5.2 ATLAS with Concurrent Queues

The next example shows how to submit multiple jobs to a concurrent queue (cf. 2.4.2). First, in lines 7 to 12, the function to execute is defined. It is supposed to just busy wait until it has received `ms_cnt` seconds of CPU time [38].

Line 19 declares and instantiates an `atlas::dispatch_queue` that is named "`a queue`" and is allowed to execute work items on CPUs `0` and `1`. Thus, the queue will create two worker threads that will execute the work items later on.

As a metric for ATLAS' predictor, we will use the amount of time that the function busy waits in seconds, set in lines 22, 25, and 28. After we have set the metric, we dispatch the work item (our function with its argument) onto the concurrent queue. Furthermore, we have to pass the deadline of this job, the metrics, number of metrics (in this case only one), and then the callable object and its arguments. As we use the `dispatch_async_atlas()` method and allow it to run on multiple CPUs, the queue may potentially allow the jobs to be executed in parallel.

In the `dispatch_*` calls, the concurrent queue will take care of predicting the execution time that is submitted to ATLAS. Also it decides on a worker thread to execute this work item. Then, using the predicted execution time, it submits the jobs to ATLAS using the `atlas_tp_submit` system call with the assigned worker thread's ID as the executing task.

When leaving the main function at line 32, the concurrent queue `q` will be destructed. `dispatch_queue`'s destructor will wait for unfinished jobs to finish.

[37] The job still preempts tasks and jobs in CFS but not jobs in the `ATLAS` band.

[38] In a working example, one would want to use `clock_gettime()` here to get the time of `CLOCK_THREAD_CPUTIME_ID`, as this clock measures the time a *thread* received.

```cpp
#include <iostream>
#include <chrono>
#include <thread>

#include "runtime/dispatch.h"

void foo(std::chrono::milliseconds dur) {
  using namespace std::chrono;
  auto end = steady_clock::now() + dur;
  while (steady_clock::now() < end)
    continue; // Busy loop
}

int main() {
  using namespace atlas;
  using namespace std::chrono_literals;
  using sc = chrono::steady_clock;

  dispatch_queue q("a queue", { 0, 1 });
  double metrics;

  metrics = 4.2;
  q.dispatch_async_atlas(sc::now() + 6s, &metrics, 1, foo, 4200ms);

  metrics = 2.3;
  q.dispatch_async_atlas(sc::now() + 12s, &metrics, 1, foo, 2300ms);

  metrics = 6.5;
  q.dispatch_async_atlas(sc::now() + 18s, &metrics, 1, foo, 6500ms);

  return 0;
}
```

Listing 2.2: Submitting an ATLAS-job using execution time prediction

3 Overload

For real-time scheduling algorithms, many properties like optimality only hold under the assumption that the system is not overloaded. For example, for EDF to be optimal, it has to be *possible* to construct a feasible schedule. In the case of overload, there is no possible schedule that allows all tasks to finish before their respective deadlines.

The utilization U_i of a periodic task τ_i is defined as the ratio of its execution time C_i divided by its period T_i: $U_i = C_i/T_i$. The utilization U_p of a system[1] of periodic tasks can be derived by summing up the utilization of all n tasks in the system: $U_p = \sum_{i=1}^{n} U_i$. If U_p exceeds 1, the schedule is definitely overloaded and no feasible schedule exists. Liu and Layland showed [LL73] that periodic task sets can be scheduled, if the system utilization does not exceed 1. In case $U_p \leq 1$, the schedulability of the task set depends not only on the parameters of the tasks but also on the used scheduling algorithm.

A system with is called overloaded, if an optimal scheduler, like EDF, does not find a feasible schedule, i.e. a schedule in which all deadlines are met. Many scheduling algorithms, like EDF, are not intended for overloaded systems. Consequently, their performance rapidly degrades under conditions of overload. Section 3.1 shows the performance of EDF under conditions of overload.

Buttazzo categorizes hard [But11] and soft [But05b] real-time scheduling algorithms into three classes, depending on how they cope with an overloaded task set:

- *Best Effort* scheduling algorithms do not reject tasks in the case of an overloaded system. The system performance can only be controlled by priority assignment. Policies for best effort scheduling were examined, for example, by Locke [Loc86].
- *Guarantee*-based[2] scheduling algorithms incorporate an acceptance test. This test verifies, whether the current task set still yields a feasible schedule when a new task is added. The acceptance test uses worst-case assumptions to determine the schedulability of the task set. If the new task set fails the acceptance test, the new task is rejected.
- *Robust* scheduling algorithms, like guarantee-based scheduling algorithms, make use of an acceptance test to determine whether enqueueing a new task still yields a feasible schedule. In addition to timing constraints, robust scheduling algorithms also take the priorities of the scheduled tasks into account. When the task set, extended by the new task, is found to not be feasible, a rejection policy is used to determine which task(s) the scheduler has to give up to maximize the system performance. By rejecting tasks based on the value of the task to the system, robust

[1] In this chapter, a system will always be a uniprocessor system.

[2] For soft real-time systems, Buttazzo calls such schedulers “scheduling algorithms with simple admission control”.

scheduling algorithms are able to achieve a better overall system performance in the case of overload. Buttazzo and Stankovic implemented a robust scheduling algorithm for sporadic hard real time tasks called Robust Earliest Deadline with single task rejection (RED) [BS93, BS95]. When a new task arrives, RED reevaluates the feasibility of the schedule. In the overloaded case, the schedule is not feasible and the system will reject the task with the least value to the system.

The cumulative value Γ_A of a scheduler A is defined as the sum over the values[3] of all tasks in the system:

$$\Gamma_A = \sum_{i=0}^{n} v(\tau_i)$$

The cumulative value of an optimal, clairvoyant scheduler is referred to by Γ^*. Using Γ_A and Γ^*, the competitive factor φ_A of a scheduling algorithm A can be calculated as

$$\Gamma_A \geq \varphi_A \Gamma^*$$

The value of φ_A is in the interval $[0; 1]$. If $\varphi_A > 0$, a scheduling algorithm A is said to be competitive. [But05b]

Baruah et al. showed [BKM+91, BKM+92], that no on-line uniprocessor task scheduling algorithm A can achieve a competitive factor φ_A greater than $1/4$ for an arbitrary task set with a loading factor greater than 2. The more general form,

$$\frac{1}{(1+\sqrt{k})^2}$$

considers the ratio k between highest and lowest value density[4]. For uniform value densities, i.e. all tasks being equally valuable to the system, the density k is 1, thus the bound for the scheduler is $1/4$.

3.1 EDF and RMS in Overload

EDF's optimality only holds under certain assumptions. One of those assumptions is, that it is possible to produce a feasible schedule. For example, with aperiodic tasks, this assumption might not hold, if jobs' deadlines are aligned such that not all jobs can meet them. A feasible schedule is a schedule in which all tasks included in the task set meet their deadlines. In the case of overload, this is not possible. As EDF is not designed to handle such situations, the performance may suffer due to the *domino effect*.

The domino effect describes a situation, in which a scheduler accepts a new task that causes a previously feasible schedule to become unfeasible. Figure 3.1[5] shows this effect for the EDF scheduling algorithm. In Figure 3.1(a), the schedule is not yet overloaded.

[3] In this context, value does not describe the result of the computation, but the value to the system as a whole, as described in Section 2.2.1.

[4] The value density of a task is the task's value to the system divided by its execution time.

[5] Release times are denoted by upward pointing triangles, deadlines by downward pointing triangles, and the time a job is executed by the length of the block.

Jobs would be scheduled in order of increasing deadline. That is, first J_2 arrives and is immediately scheduled for execution. While J_2 executes, J_3 and J_4 arrive. As J_3's and J_4's deadlines are later than the deadline of J_2, EDF does not need to preempt J_2. Next, J_1 arrives. As J_1's deadline is earlier than that of J_2, J_2 is preempted and J_1 executed instead. After J_1 finishes, J_2 is continued till completion. Finally, J_3 and J_4 are executed in full.

As EDF itself does not incorporate an acceptance test, new tasks are always accepted the system when they arrive. This can be seen in Figure 3.1(b): J_0 arrives while J_1 is still executing. As the deadline of J_0 is earlier than J_1's deadline, J_0 preempts J_1 and is executed to completion. After that, all other jobs miss their deadlines.

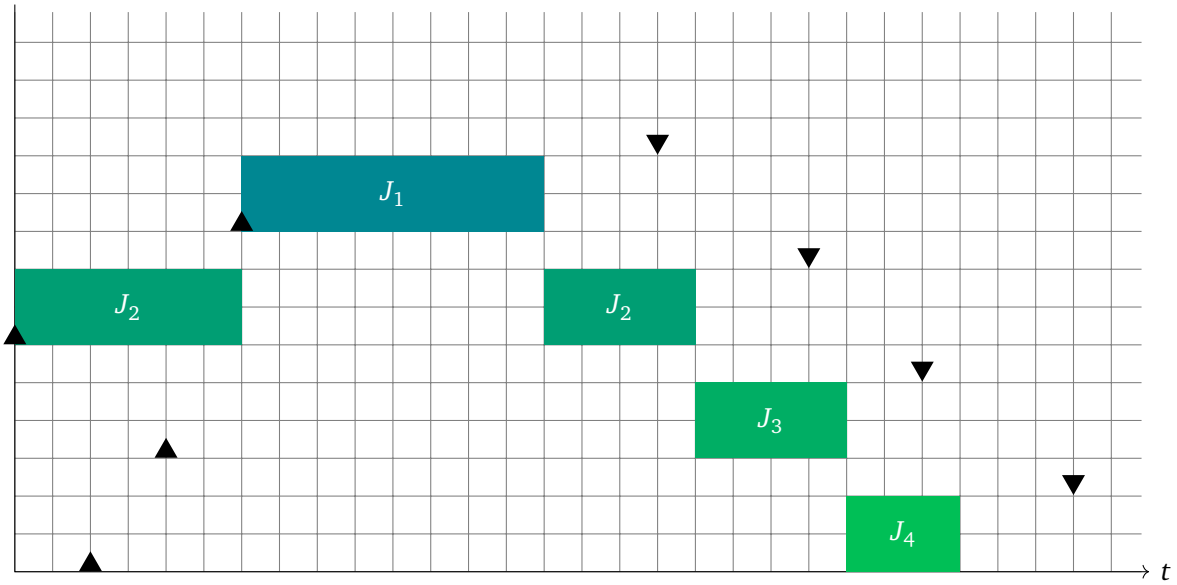
Buttazzo compared the EDF algorithm to fixed priority scheduling with rate monotonic priority assignment (better known as RMS) for overloaded systems of periodic tasks. In case the system was permanently overloaded, tasks in EDF behave as if they were executed at a lower rate. In RMS, tasks of lower priority, i.e. tasks with longer periods, get blocked under overload. Though both scheduling algorithms behave predictably, which one is preferable is application dependent.

Transient overload describes a situation, in which a system is overloaded for a certain amount of time, but may recover and transition to a state of being non-overloaded again. In transient overload, any task in EDF may miss its deadline. In RMS, all tasks except the highest priority one may miss their deadlines. Though this may seem to offer an advantage over EDF, in practice, for most applications this is not relevant. [But05a]

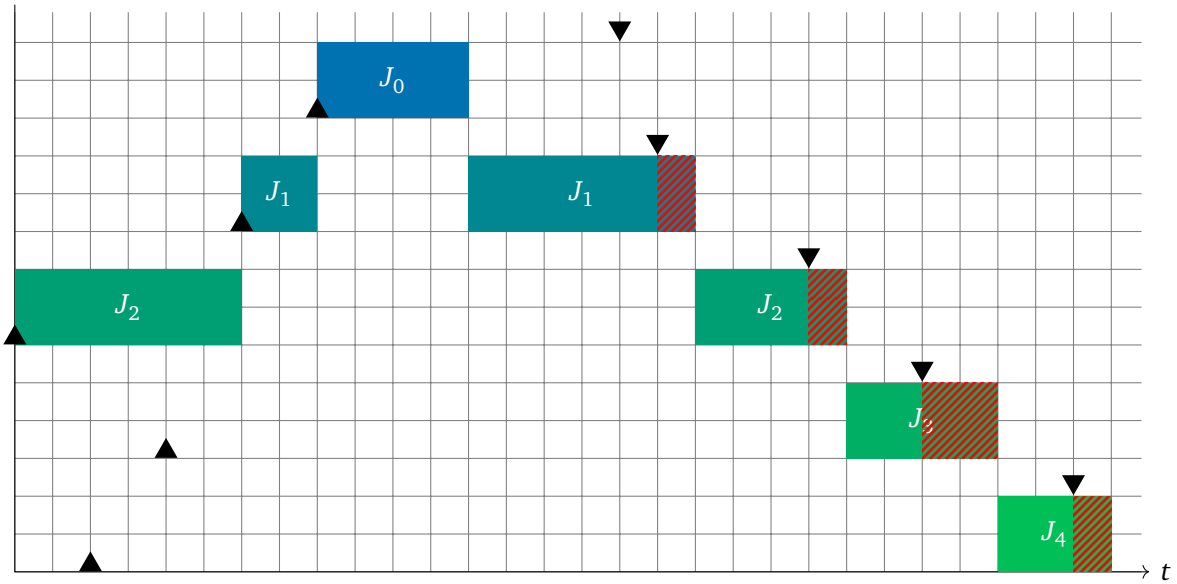
Due to EDF's sub par performance under overload, Buttzzo and Stankovic implemented different schedulers [BS93] that are conceptually derived from EDF but incorporate acceptance and/or rejection policies. Tasks in the following models are described by the tuple $J_i = (C_i, D_i, m_i, v_i)$, where C_i and D_i denote the WCET and deadline of the task respectively. The value m_i represents the task's deadline tolerance and v_i the explicitly stated importance of the task J_i, relative to other tasks in the task set. By using the deadline tolerance m_i, the scheduling algorithms presented here compensate for the pessimistic estimate by the WCET. The following scheduling algorithms split real-time tasks into the classes *HARD* and *CRITICAL*. *HARD* tasks are guaranteed to complete in a non-overloaded system. *CRITICAL* tasks are ones that are guaranteed at runtime, that a certain portion of tasks will finish within their deadline:

- The Guaranteed Earliest Deadline (GED) algorithm uses an acceptance test to verify before accepting a new task, whether the new task set is schedulable. If it is not schedulable, the new task is rejected.

- The Robust Earliest Deadline with single task rejection (RED) algorithm, like GED, uses an acceptance test to ensure, that the new task set is schedulable. If it is, the new task gets accepted. Otherwise, RED finds the task with the least value to the system and tests, whether the task set would be schedulable if the task with the least value was rejected. In this case, the least value task is rejected and the new task accepted. If not, the new task is rejected. RED also keeps a list of rejected tasks. When a task finishes, the previously rejected task that can still meet its deadline with the greatest value may be reaccepted.



(a) A non-overloaded schedule



(b) Job J_0 is accepted, causing the domino effect

Figure 3.1: The domino effect [But05b]

- The Robust Earliest Deadline with multiple task rejection (MED) algorithm is the same as the RED algorithm, but might reject more than one lower value task, if the new task is critical.

3.2 TAFT scheduler

The Time-Aware Fault-Tolerant (TAFT) scheduler [BG01, GN02] is based on Version 3.0 of RTLinux. Tasks in TAFT are defined as a TaskPair (TP) that comprises two parts: the MainPart (MP) and the ExceptionPart (EP). The MP represents the regular execution of the program, in which execution times might not be known a priori. Thus, the MP is characterized by the expected case execution time (ECET) which is comparable to ATLAS' execution time prediction[6]. An MP's ECET may be found by online analysis of the program. The EP represents a critical component of a TP, characterized by a WCET. TAFT is a hard real-time scheduler in the sense that it guarantees either one of the components to complete within the given deadline of a TP.

The scheduler treats the MP and EP as separate scheduling entities. MPs and EPs are scheduled in different scheduling bands. The scheduling band for EPs uses an LRT approach to schedule EPs at the latest point in time possible. MPs are scheduled using the EDF scheduling policy. If an MP exceeds its ECET, its priority is set lower than that of all other MPs to avoid a domino effect that is inherent when scheduling with EDF otherwise. Whenever an EP becomes runnable, it preempts any MP that might be running. Also the MP, that corresponds to this EP, is removed from the schedule. By this scheduling strategy, an MP is always preempted by its EP, if the MP would not finish before the EP needs to begin execution.

For the TAFT scheduler, a TP τ_i is defined as a tuple (T_i, D_i, C_i, E_i) with the activation period T_i, the deadline D_i, the ECET C_i of the MP, and the WCET E_i of the EP.

For accepting new tasks, the TAFT scheduler uses an acceptance test, that works on an ordered set Π of n tasks $\left\{\tau_i, i = 1 \text{ to } n, \forall i, j : \left(T_i \leq T_j \Rightarrow T_i | T_j\right)\right\}$. The acceptance test employs the maximum utilization factor (MUF) Ω_i for each task τ_i. Ω_i is defined as

$$\Omega_i = \sum_{j=1}^{i} \frac{C_j + E_j}{T_j} + \frac{1}{T_i} \sum_{i<j\leq n} E_j$$

The TAFT scheduler can schedule a set Π if $\Omega_i \leq 1$ for all tasks τ_i.

3.3 ROBUST

The Resistance to Overload By Using Slack Time (ROBUST) scheduling algorithm [BH93] handles tasks with hard deadlines on uniprocessor systems in an on-line and preemptive fashion.

[6] Except, that the ECET is not automatically predicted.

Tasks in ROBUST are defined as a set of arrival time $T.a$, relative deadline $T.d$, and execution time requirement $T.e$. $T.e_r$ denotes the remaining execution time the task requires. The slack factor of a task T is defined as $T.d/T.e$.

Baruah and Haritsa measure the performance of a system using the Effective Processor Utilization (EPU). The EPU in an interval $[t_s; t_f[$ is defined as:

$$EPU = \frac{\sum_{i \in C} x_{i_{[t_s;t_f[}}}{t_f - t_s}$$

Here, C is the set of all tasks in the system and $x_{i_{[t_s;t_f]}}$ denotes the amount of processor time received by task i in the interval $[t_s; t_f[$, *if it finishes* within the interval.

The ROBUST scheduling algorithm is able to achieve an EPU of at least 50 % if all submitted tasks are guaranteed to have a slack factor of at least 2. In contrast, other schedulers without this restriction can at most achieve an EPU of 25 % [BKM+91, BKM+92].

In an overloaded interval, ROBUST schedules tasks by dividing the interval into $2i$ phases (Phase-1 to Phase-$2i$). Each even-numbered phase's length is the same as the length of the preceding (odd-numbered) phase. Scheduling is handled differently, depending on whether currently an even- or odd-numbered phase is executed:

- odd-numbered phase $(2i-1)$: The task with the longest execution time requirement $T_{max}.e$ is selected and scheduled non-preemptively. If a new task T_{new} arrives in this phase, two possibilities exist:

  1. $T_{new}.e < T_{max}.e$: The new task's execution time requirement is less than that of the currently scheduled task. Thus, ROBUST values it less.

  2. $T_{new}.e \geq T_{max}.e$: Due to the requirement of tasks' slack factor being at least 2, the new task can still be scheduled in the next phase.

- even-numbered phase $(2i)$: Again, the task with the longest execution time requirement $T_{max}.e$ is selected and executed preemptively. If a new job T_{new} arrives with an execution time requirement $T_{new}.e$ that is longer than $T_{max}.e$, T_{max} is preempted and T_{new} is executed instead.

After an even-numbered phase, the task that was executed last is T_{max} for the next odd-numbered phase. The length of the next phase and the phase after that is set to $T_{max}.e_r$ and the task is scheduled non-preemptively in the odd-numbered phase.

Using this scheduling algorithm, ROBUST can guarantee that the processor is utilized with a task that finishes in each odd-numbered phase. Depending on whether jobs finish in the following even-numbered phase, the EPU might increase, thus

$$\text{EPU} \geq \frac{\sum_{j=1}^{i} [\text{ length of Phase-}(2j-1)\text{ }]}{\sum_{k=1}^{2i} [\text{ length of Phase-}k\text{ }]} = \frac{1}{2}$$

holds.

3.4 D^{over}

With D^{over}, Koren and Shasha presented an on-line scheduler [KS92], that reaches the best possible bound for the competitive factor for arbitrary task sets of $\frac{1}{(1+\sqrt{k})^2}$. It schedules firm and soft real-time tasks preemptively.

When the system is underloaded, D^{over} schedules tasks like EDF would. Tasks are split into sets: *privileged* and *waiting* tasks. Whenever a task is preempted, it becomes privileged. A task's Latest Start Time (LST) can be calculated by subtracting its remaining computation time from the task's deadline.

In case the system is overloaded, D^{over} schedules in a different manner: When a task reaches its LST, D^{over} checks, whether

$$v_{next} > \left(1 + \sqrt{k}\right)\left(v_{current} + V_{priv}\right)$$

where v_{next} is the value of the task reaching its LST, $v_{current}$ the value of the task that is currently being executed, and V_{priv} the sum over all values of privileged tasks. If the comparison evaluates to true, the LST-task is executed and all other tasks become waiting tasks. Otherwise, the task is abandoned.

3.5 CBS

Abeni and Buttazzo [AB98] introduced CBS as a method for scheduling soft real-time tasks alongside hard real-time tasks. The work combines the concepts of the Dynamic Sporadic Server (DSS) and the Total Bandwidth Server (TBS) while aiming to enhance the performance of the system in overload situations by reclaiming unused CPU time caused by early completions for jobs of the same task. It distinguishes two types of tasks[7]:

- *Hard real-time tasks* are scheduled using the EDF algorithm based on their WCET and period.
- *Soft real-time tasks* are assigned to a CBS and scheduled using their mean execution times instead of the WCET and the desired activation period.

For CBS, a hard real-time task is defined as $\tau_i = (C_i, T_i)$ where C_i is the WCET of each job and T_i is the task period. For soft real-time tasks, C_i denotes the mean execution time of jobs and T_i is the desired activation period. Each task τ_i emits jobs $J_{i,j}$ that are associated with the following parameters: arrival time $r_{i,j}$, finishing time $f_{i,j}$, and dynamic deadline $d_{i,j}$. The dynamic deadline is assigned to the task by a CBS instance[8].

[7] Actually, CBS distinguishes three types of tasks: hard, soft, and non real-time tasks. However, throughout the paper, non real-time tasks are not mentioned anymore.

[8] Note, that CBS is used in two different contexts here: on the one hand as the concept of CBS and on the other hand as concrete CBS instances in the system. I will call the concept always just "CBS" and the instance "CBS instance".

A CBS instance s is defined by $s = (Q_s, T_s)$ where Q_s is the maximum server budget and T_s is the server period. The server bandwidth is calculated as $U_s = Q_s/T_s$. At runtime, each CBS instance holds the parameters c_s and $d_{s,k}$. c_s is the server's remaining budget and $d_{s,k}$ the k-th deadline of the server.

CBS utilizes a global EDF schedule, that schedules hard real-time tasks that are not handled by a CBS instance and all CBS instances that are in the system. A CBS instance in turn can schedule jobs of either one hard real-time task or multiple soft real-time tasks.

When scheduling jobs of a task that is assigned to a CBS instance, the CBS instance reacts to different types of events:

- *When the server budget reaches zero*, it is replenished to the maximum budget Q_s and a new server deadline is generated that is one server period T_i later than the last one.
- *When a job is executed*, the server budget is decreased by the corresponding amount of time.
- *When a new job arrives*, it is enqueued in the server queue.
- *When a job finishes*, it is dequeued and the next job in the queue is served.
- *When a job is served*, the server decides, whether the remaining server budget is sufficient to serve that job. If it is, it is executed using the old server deadline and budget. Otherwise, a new deadline is generated that is one server period T_i later than the current one and the budget is restocked to the maximum budget Q_s.

CBS is able to give different guarantees. The *isolation property* holds for a system scheduling multiple hard real-time tasks and a CBS instance. It guarantees that iff. $U_p + U_s \leq 1$ holds[9], all hard real-time tasks and the CBS instance can be scheduled. Consequently, if the parameters of the CBSes instance are chosen such that the condition holds, soft real-time task cannot influence the scheduling of hard real-time tasks and *temporal isolation* exists between hard real-time tasks and the CBS instance.

If a hard real-time task $\tau_i = (C_i, T_i)$ is scheduled by a CBS instance with parameters (Q_s, T_s) and the parameters are chosen such that $C_i \leq Q_i$ and $T_i = T_s$ holds, the CBS instance schedules the hard real-time task just as the EDF algorithm would.

If the distribution of computation times over the set of jobs of a soft real-time task is known, CBS is able to give statistical guarantees for jobs, i.e. the probability of jobs meeting their deadlines.

3.6 Adaption to Overload

In "Soft Real-Time Systems: Predictability vs. Efficiency", Buttazzo describes [But05b] multiple ways for systems to cope with overload, if the system does not want to, or cannot, reject incoming tasks:

[9] U_p is the processor utilization of the hard real-time tasks and U_s the bandwidth of the CBS instance.

- *Service adaption*
- *Job skipping*
- *Period adaption*

3.6.1 Service Adaption

Some algorithms allow the results of computations to be refined the longer that algorithm runs. This allows a scheduling algorithm to trade precision for computation time in the case of overload. Lin et al. [LNL87] explored this possibility for the Concord project and generalized the concept of imprecise calculations. Buttazzo proposes a model for imprecise calculations, that splits each task J_i into a mandatory subtask M_i and an optional subtask O_i with their respective computation times m_i and o_i. The arrival time a_i and deadline d_i are the same for both subtasks. The mandatory subtask must be executed and calculates a minimal result. This result is then improved by the optional part if it is scheduled. The longer the optional part is executed, the more precise the result becomes. For such a task model, a suitable metric is to use the average error

$$\bar{\epsilon} = \sum_{i=1}^{n} w_i(o_i - \sigma_i)$$

where w_i denotes the relative importance of J_i and σ_i the processor time allocated to O_i. In a system that schedules tasks split in such a way, a schedule is called *feasible*, if every mandatory subtask M_i is completed within its deadline. If all optional parts are executed in full, i.e. the average error $\bar{\epsilon}$ is 0, the schedule is said to be *precise*.

For tasks that cannot be split into a mandatory and an optional subtask, Buttazzo proposes supplying multiple versions of the task. Here, higher quality versions and thus longer ones may be replaced by the scheduler with lower quality ones in case of overload.

3.6.2 Job Skipping

For periodic task sets, Koren and Shasha presented a model [KS95], in which the scheduler is allowed to skip a certain amount of jobs. By not having to execute every job of a task, the scheduler gains flexibility to better cope with overload. In this model, task definitions include a *skip parameter* s_i with $2 \leq s_i \leq \infty$ that gives the minimum distance between skipped jobs. Every job is either red or blue. A red job is mandatory and needs to be executed in full. Blue jobs may be aborted at any point in their execution if the need arises. If a blue job is aborted, the next job must be red, as the minimum inter-skip distance is 2. If a blue job completes, the next job will also become a blue job, as the job was not aborted. Koren and Shasha propose two on-line scheduling algorithms:

- Red Tasks Only (RTO)[10] schedules red jobs using the EDF algorithm and always

[10]The term “task” here comes from the fact, that Koren and Shasha talk about task instances instead of jobs.

rejects, i.e. aborts, blue tasks.

- Blue When Possible (BWP) also schedules red jobs using the EDF algorithm but allows blue jobs to run, when no red job is runnable. As the overall system performance is calculated based on all jobs, BWP performs better than RTO according to this metric.

Buttazzo extended this concept by using a CBS (cf. Section 3.5) to utilize the computation time that might be gained by skipping blue jobs when RTO is used. The CBS uses its computation time to advance aperiodic jobs. Koren and Shasha showed that the problem of determining, whether a set of periodic, skippable jobs is schedulable by EDF is NP-hard.

For a given task set $\Gamma = \{T_i(p_i, c_i, s_i)\}_{i=1}^{n}$ of n tasks with period p_i, execution time c_i, and skip parameter s_i to be feasibly schedulable,

$$L \geq \sum_{i=1}^{n} D\left(i, [0, L]\right)$$

has to hold for all $L \geq 0$. L represents an arbitrary point in time. The equation checks whether the schedule requires less time than available within the interval $[0; L]$. In this equation, the required time of a task T_i in the interval $[0, L]$ is defined by

$$D\left(i, [0, L]\right) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i$$

From this, the definition of the processor utilization follows:

$$U_p^* = \max_{L \geq 1} \left\{ \frac{\sum_i D(i, [0, L])}{L} \right\}$$

If a CBS is used in conjunction with RTO, a sensible schedulability test might be

$$U_p^* + U_s \leq 1$$

where U_s denotes the utilization of the CBS as described in Section 3.5.

3.6.3 Period Adaption

Another option for handling overload with periodic task sets is the “elastic model”. In this model, the periods of tasks are considered flexible within a certain range. The model is adapted from a spring model, that augments each task by a minimum and a maximum period as well as an elasticity coefficient. Tasks are described by:

- C_i computation time
- T_{i_0} minimum period
- $T_{i_{max}}$ maximum period
- E_i elasticity coefficient

The elasticity coefficient describes the ability of the task to adapt its period at runtime. An elasticity coefficient of 0 means that the scheduler may chose an arbitrary but fixed period within the interval $\left[T_{i_0}; T_{i_{max}}\right]$. If E_i is greater than 0, the period may change. The relation of different tasks' elasticity coefficients determines, how much a task's period changes, when the load situation in the system changes. The greater E_i is, the more the task can change. A periodic hard real-time task may be modelled by choosing T_{i_0} equal to $T_{i_{max}}$.

4 Design & Implementation

To cope with overloaded situations, I extended the kernel part of ATLAS as well as the userland part. The kernel part was implemented in C, while the userland part uses C++14 with help from the Standard Template Library (STL). In the following I will first describe the kernel component, design decisions and the different methods for detecting and handling overload in ATLAS. Thereafter, the load manager, which is implemented in userland, will be presented.

4.1 Formalism

To ease the description of the different policies used, this section introduces some formalisms that are used throughout this chapter.

An ATLAS-job J_i is defined by the tuple (e_i, d_i). The parameter e_i denotes J_i's registered execution time, while d_i represents the absolute deadline of the job. At runtime, the job furthermore holds parameters $\hat{e}_i$, $\hat{d}_i$, and r_i. The parameters $\hat{e}_i$ and $\hat{d}_i$ represent the scheduled execution time and scheduled deadline respectively. The received execution time of a job is given by r_i.

The current point in time will be denoted by t. Using it, the slack s_i of a job J_i is defined as $s_i = \hat{d}_i - r_i$.

An ATLAS-run queue R_j is defined as a set of n_j ATLAS-jobs scheduled on it. The set is ordered by the value of $\hat{d}_i$.

From the current point in time t, the available time on the run queue can be calculated as

$$a_j = max_{J_i \in R_j}(d_i) - t$$

and the run queue demand as

$$w_j = \sum_{J_i \in R_j} (e_i - r_i)$$

.

The required cutback of a run queue R_j is denoted by c_j, and calculated using

$$c_j = w_j - a_j$$

```
...
[function modifies atlas_job_tree]
...
if (is_overloaded(rq))
        handle_overload(rq);
...
```

Listing 4.1: Calls to `is_overloaded` and `handle_overload`

4.2 Kernel

The kernel component of ATLAS was extended by facilities for detecting and handling overload. For this, the functions `is_overloaded` and `handle_overload` were implemented.

- `is_overloaded(struct atlas_rq *rq)` checks, whether the given ATLAS run queue is overloaded. It does so using different methods that are shown in Section 4.2.1.
- `handle_overload(struct atlas_rq *rq)` assumes there is overload and modifies the scheduled execution times of jobs on the `ATLAS` run queue to transition the schedule into a state of being non-overloaded again.

Earlier versions of overload management for ATLAS looked at all places where the ATLAS schedule, represented by the structure `atlas_job_tree`, is modified. Namely these are:

- `insert_job_into_tree`: Here new jobs get inserted into the corresponding ATLAS job tree. This might happen, when a new job arrives, but also when a job is moved from, for example, the `ATLAS` to the `CFS` scheduling band.
- `remove_job_from_tree`: Similar to `insert_job_into_tree`, this function is called when a job either changes scheduling policies within ATLAS or finishes and is thus removed from the `atlas_job_tree`.
- `update_curr_atlas`: This function is ATLAS' implementation of the function pointer `update_curr` in the structure `sched_class`. As such, it is regularly called when the scheduler timer expires and updates statistics of the currently running job. As this might modify the schedule, I also check here for overload and handle it if necessary.

All calls in those functions looked basically like shown in Listing 4.1. First, the implementation checked for overload, and after that handled the overload if needed.

To allow the user to easily disable overload management for ATLAS, a design like in Listing 4.1 allows the programmer to just surround the code snippet with `#ifdef`. If no overload management is needed, there exists no runtime overhead. Also all of the code for overload management[1] is contained in `atlas_overload.h` and `atlas_overload.c`.

[1] All except the calls to `is_overloaded`, `handle_overload` and the system call `get_load_stats`.

So, when ATLAS' overload management is not built into the kernel, the code also uses no space in memory.

Unfortunately, this approach proved too unstable to be usable. The overload handling only considers jobs in the `ATLAS` band. As during the execution of the schedule, jobs are moved between scheduling bands, the schedule looks less loaded, when jobs have already moved to the `EDF Recovery` or `CFS` band. This caused the overload handling to assign the remaining jobs a larger portion of the remaining time than originally intended. Thus, only the call to `handle_overload` remained in `insert_job_into_tree`. This call is only performed, if the `atlas_job_tree` that the job is inserted into is the one used by the `ATLAS` band.

Though `is_overloaded` is used no more from the scheduler part of the ATLAS kernel, its results are still available from userspace via the `atlas_getload` system call.

4.2.1 Overload Detection

To evaluate the load situation of the system, two methods are available. Both were proposed by Weisbach [Wei16]. These methods are:

- *Slack*-based overload detection uses the slack of the first job in the schedule of ATLAS. If this slack is negative, overload exists and also the amount of time that is needed in the schedule is given by the negative slack.
- The *Interval Load Function* gives a measurement of how much time is used by the schedule in a given window. In the current implementation it can be used in two different modes. In the *Interval Load Function* mode, the window size is fixed and can be configured via sysctl[2]. In the *Horizon Load Function* mode, the window is automatically set to the time span from the current time to the latest deadline in the schedule (a.k.a. the horizon).

The different types of overload detection can be selected by the privileged user via sysctl. The sysctl-variable to control, which overload detection was used is called `kernel.sched_atlas_overload_detection`. Possible values for this variable are listed in Table 4.1.

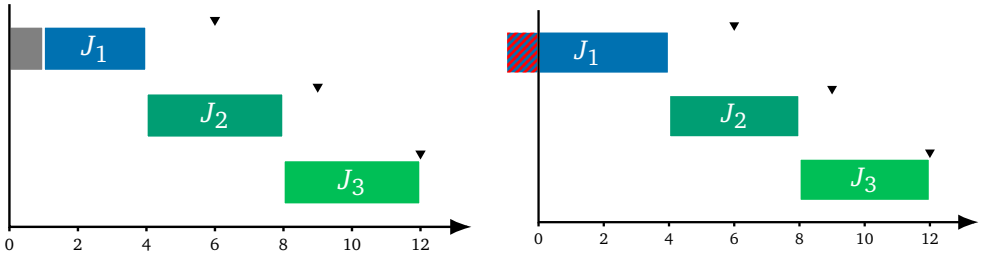
Detection name	Value	Overload Detection
`OL_DET_NONE`	0	No overload detection
`OL_DET_SLACK`	1	Slack-based
`OL_DET_ILF`	2	Interval Load Function-based
`OL_DET_HLF`	3	Horizon Load Function-based

Table 4.1: Possible values for `kernel.sched_atlas_overload_detection`

[2]For specific configuration variable names and defaults, see Section 4.2.3.

The Figures 4.1(a) and 4.1(b) show a schedule both in a non-overloaded and in an overloaded situation. In Figure 4.1(a), the jobs are arranged in the schedule in order of increasing deadline with the last job finishing exactly at its deadline. Each job's starting time is the previous job's finishing time. The first job still has one time unit left until it needs to start. When scheduling with ATLAS this time is used to allow the job to pre-roll and by that possibly reduce the time, the job needs to execute in the `ATLAS` band.

If job J_1 had reserved 5 time units of execution time instead of 3, the schedule would be overloaded. This situation is depicted in Figure 4.1(b), where job J_1 has a negative slack of -1.



(a) Non-overloaded schedule

(b) Overloaded schedule

Figure 4.1: Slack-based overload detection

An interval is said to be tightly-packed, if at every time in this interval, a job is scheduled to execute, i.e. there exist no gaps in the schedule.

To reliably detect a situation, in which the schedule is overloaded, using the slack of the first job is sufficient. ATLAS builds its schedule by aligning the jobs' scheduled finishing times with their deadline or the latest release time of the following job, depending on which one is earlier.

If the remaining execution time of a job is greater than the time until its deadline, this job cannot finish in time and all jobs in the interval from this job to the end of the tightly-packed interval are in the overloaded interval.

Overload at a point in time later than the first overloaded interval cannot exist. If it would, jobs would be moved closer to the overloaded interval and eventually reach it, thus also belonging to the tightly-packed and overloaded interval at the beginning.

Consider the example in Figure 4.1. If there were further jobs $J_4 = \left(e_4 : 1,\ d_4 : 14\right)$ and $J_5 = (e_5 : 2,\ d_5 : 15)$, J_4 would be scheduled to start at time 13 and J_5 at 14. The system would still be overloaded due to the jobs J_1 to J_3. If e_4 and e_5 would be set to 2 instead, they would reach the first overloaded interval (including jobs J_1 to J_3) and move it to an earlier starting point. This would only increase the negative slack of job J_1, but not produce an overloaded interval at a later point in time.

Interval/Horizon Load Function-based Overload Detection

The Interval Load Function (ILF) and the Horizon Load Function (HLF) both give a measurement of how utilized an interval is. As I only use the ILF with an interval starting at the current point in time t until an arbitrary point in time, u, and only consider jobs, the definition can be simplified from the definition given by Weisbach [Wei16]:

$$l_i(u) = \begin{cases} e_i - r_i & \hat{d}_i \leq u \\ 0 & \text{otherwise} \end{cases}$$

This definition can be extended to run queues as

$$\sum_{J_i \in R_j} l_i(u)$$

The interval $[t;u]$ used by the ILF is bound by the current point in time and the time u. For the ILF, the window size can be configured. The window used by the HLF is determined by the latest deadline in the schedule.

As the ILF and HLF are not able to detect overload, they might only be used to detect whether enqueueing a new job would overload the schedule. Even this cannot be done reliably with the ILF/HLF, as false-positives might occur, when there are gaps in the schedule. However, they could be used as an admission criterion when submitting jobs. As ATLAS currently does not reject jobs, I did not examine these options any further. Instead, the HLF is used with the load manager to determine how much time is available in the schedule.

4.2.2 Overload Handling

As explained in Section 4.2, the function `handle_overload` is called to transition the ATLAS run queue back into a state of being non-overloaded. To achieve this, first the required cutback has to be determined. Then, the cutback is applied to a run queue using a given policy. The policy determines how the cutback is distributed among all jobs on the run queue.

When the required cutback has been determined, it is applied to the run queue by a given policy. The available policies are listed in Table 4.2 and described next. If no overload handling is needed or wanted, `OL_POL_NONE` should be chosen. In this case, the `handle_overload` function is still called but returns early without calculating the required cutback.

The available policies are:

- *Equal cutback*: By using the policy `OL_POL_FIXED`, the required cutback

$$c_j = w_j - a_j$$

Handling Name	Value	Overload Handling
`OL_POL_NONE`	0	No overload handling
`OL_POL_FIXED`	1	Fixed cutback
`OL_POL_PROP`	2	Proportional cutback
`OL_POL_LAXITY`	3	Laxity-proportional cutback
`OL_POL_FAIR`	4	Cutback to fair share
`OL_POL_DROP`	5	Cutback is applied to the last job

Table 4.2: Possible values for `kernel.sched_atlas_overload_policy`

is split equally among all jobs on the run queue. This allows for an easy calculation of the per-job-cutback, but has the drawback, that short jobs are proportionally cut back more than longer jobs. Each job's cut back is thus

$$\bar{c}_j = \frac{c_j}{n_j}$$

and the scheduled execution time of each job calculates as

$$\hat{e}_i = \begin{cases} e_i - \bar{c}_j & \text{if } \bar{c}_j < e_i \\ 0 & \text{otherwise} \end{cases}$$

- *Proportional cutback*: This policy avoids the previous jobs drawback by distributing the required cutback proportionally to the scheduled execution time of each job. The drawback in this case is, that jobs still can register more execution time than they need. Therefore, it should only be used in a system in which the submitted execution times relate to the actual need of the job, for example when using ATLAS with its predictor. A job's scheduled execution time is set to

$$\hat{e}_i = \frac{e_i \cdot a_j}{\sum_{J_k \in R_j} e_k}$$

  i.e. the job receives an amount of execution time that is directly proportional to the fraction of the job's submitted execution time and the sum over all execution times.

- *Laxity-based cutback*: For transient overload, it might be beneficial to cut back later jobs more than earlier ones, as the load might still decrease while the jobs in the schedule are executed. This might happen due to jobs finishing early or because of applications that are able to adapt in overload and reduce the load they generate by using the `atlas_update()` and `atlas_remove()` system calls. For this situation, `OL_POL_LAXITY` cuts jobs back proportional to their laxity. A job J_i's laxity at time t is defined as

$$\text{laxity}_i(t) = \begin{cases} \text{slack}_i(t) & \text{if slack}_i(t) > 0 \\ 0 & \text{otherwise} \end{cases}$$

  The slack of a job J_i in turn is given by

$$\text{slack}_i(t) = (d_i - (e_i - r_i)) - t$$

Jobs with a high laxity get cut back more, while jobs that have little or no laxity get cut back less or not at all. For each job, the scheduled execution time is calculated by

$$\hat{e}_i = e_i - \frac{c_j \cdot \text{laxity}_i(t)}{\sum_{J_i \in R_j} \text{laxity}_i(t)}$$

If $\hat{e}_i$ would be set to a negative value, it is set to 0 instead.

- *Cutback to fair share*: This policy guarantees each job a minimum amount of execution time in the `ATLAS` scheduling band of ATLAS. Jobs' scheduled execution times are set to the fair share that is available until the latest deadline in the schedule. This is currently implemented by just dividing the time until the latest deadline by the number of jobs. As jobs might have reserved less time than the fair share, the difference should be available for other jobs. In the performed experiments, this does not pose a problem, as the jobs that work less than the fair share donate this time as slack to other jobs. A job's scheduled execution time is calculated the following way:

$$\hat{e}_i = \frac{a_j}{n_j}$$

- *Cutback last jobs*: The `OL_POL_DROP` policy applies the whole cutback to the last job, if its execution time is long enough. Otherwise the job's scheduled execution time is set to 0, effectively scheduling it in the `EDF Recovery` band of ATLAS. In case the required cutback is longer than the last job's scheduled execution time, the remaining required cutback is applied to the next to last jobs, until the required cutback is distributed. Let

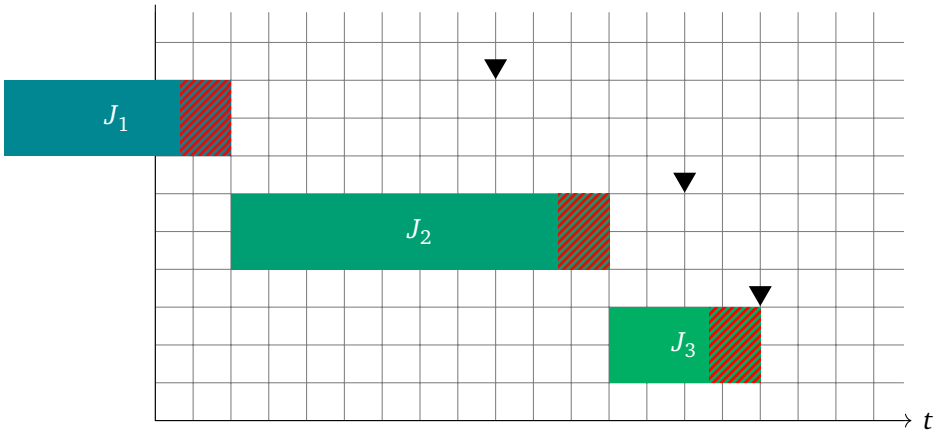
$$sx(i) = \sum_{k=i}^{n_j} e_k$$

be the sum over all submitted execution times from job J_i to the end of the schedule. Then, a job's scheduled execution time is calculated as:

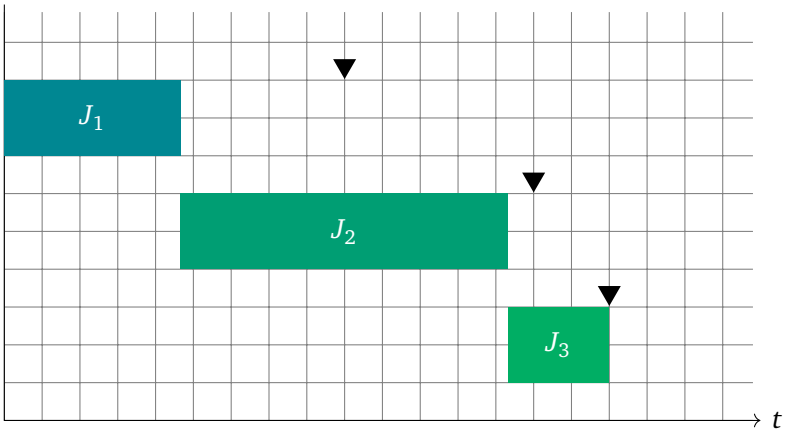
$$\hat{e}_i = e_i - \begin{cases} e_i & c_j \geq sx(i) \\ c_j - sx(i+1) & sx(i) > c_j > sx(i+1) \\ 0 & sx(i) \geq c_j \end{cases}$$

The implementation for this policy iterates over all jobs in reverse order. At each job, the algorithm checks, if the remaining required cutback is larger than the submitted execution time of this job. If it is, this jobs scheduled execution time is set to zero and the submitted execution time subtracted from the required cutback. If it is not, the jobs scheduled execution time is set to the difference between its submitted execution time and the required cutback and the required cutback to zero. After the required cutback is set to zero, the algorithm sets the scheduled execution time of jobs to their submitted execution time.

The next sections show examples of how the different scheduling policies apply a required cutback of 2 time units to a given schedule. The current time t is always 0 for these examples.



(a) An overloaded schedule

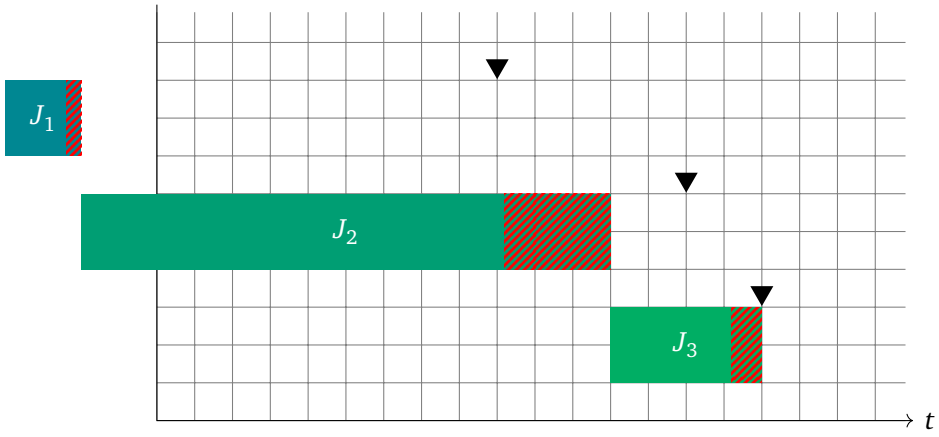


(b) The non-overloaded schedule after applying a fixed cutback to it

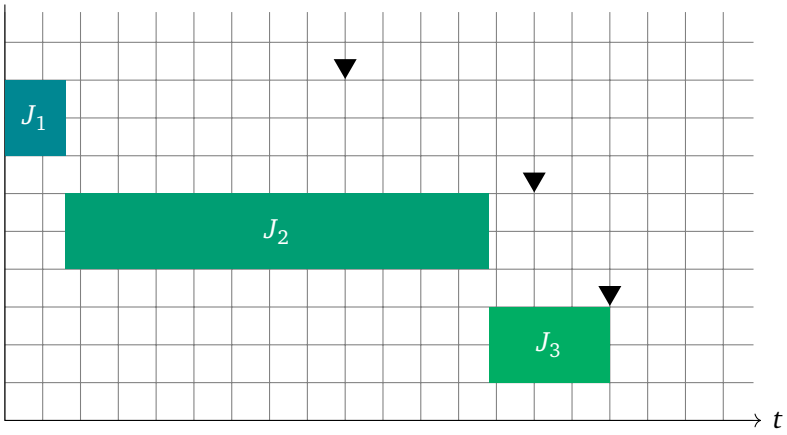
Figure 4.2: Application of a fixed cutback to the schedule

Figure 4.2(a) shows a system that has 3 jobs $J_1 = (e_1 : 3,\ d_1 : 4.5)$, $J_2 = (e_2 : 5,\ d_2 : 7)$, and $J_3 = (e_3 : 2,\ d_3 : 8)$. The resulting schedule is overloaded by two time units that are distributed among the jobs using the policy `OL_POL_FIXED`. Each job's portion of the cutback, in the image shown as the red pattern, on the run queue R_j is c_j/n_j. In this case, each per-job-cutback is $2/3$ time units, resulting in scheduled execution times of $\hat{e}_1 = 7/3$ for job J_1, $\hat{e}_2 = 13/3$ for job J_2, and $\hat{e}_3 = 4/3$ for job J_3. The schedule after applying the cutback to all jobs is depicted in Figure 4.2(b).

Proportional cutback



(a) An overloaded schedule



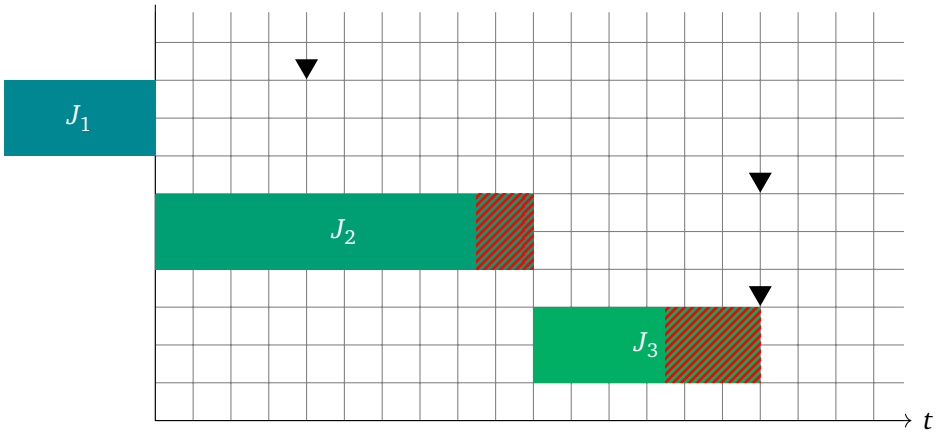
(b) The non-overloaded schedule after applying a proportional cutback to it

Figure 4.3: Application of a proportional cutback to the schedule

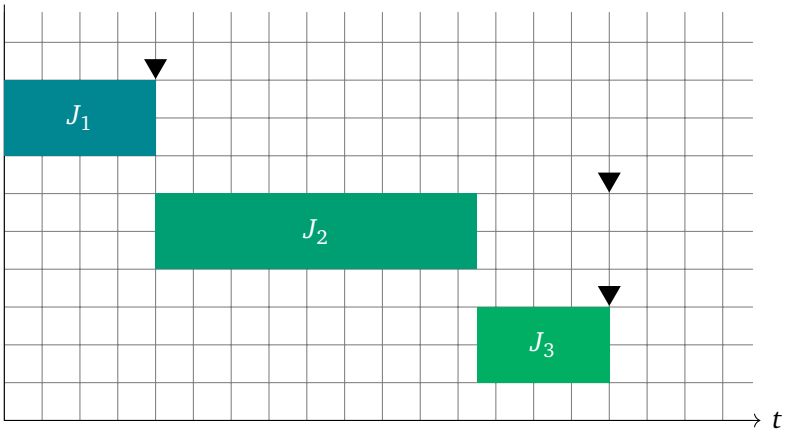
The schedule is again overloaded by two time units in Figure 4.3(a). The jobs present in the schedule are $J_1 = (e_1 : 1,\ d_1 : 4.5)$, $J_2 = (e_2 : 7,\ d_2 : 7)$, and $J_3 = (e_3 : 2,\ d_3 : 8)$. Each job's cutback on run queue R_j is calculated as

$$\frac{e_i \cdot a_j}{\sum_{J_k \in R_j} e_k}$$

Thus, J_1's resulting scheduled execution time $\hat{e}_1$ is $^8/_{10}$, $\hat{e}_2$ is $^{56}/_{10}$, and J_3 receives a scheduled execution time of $\hat{e}_3 = {}^{16}/_{10}$. Figure 4.3(b) shows the non-overloaded schedule after applying the respective cutbacks.



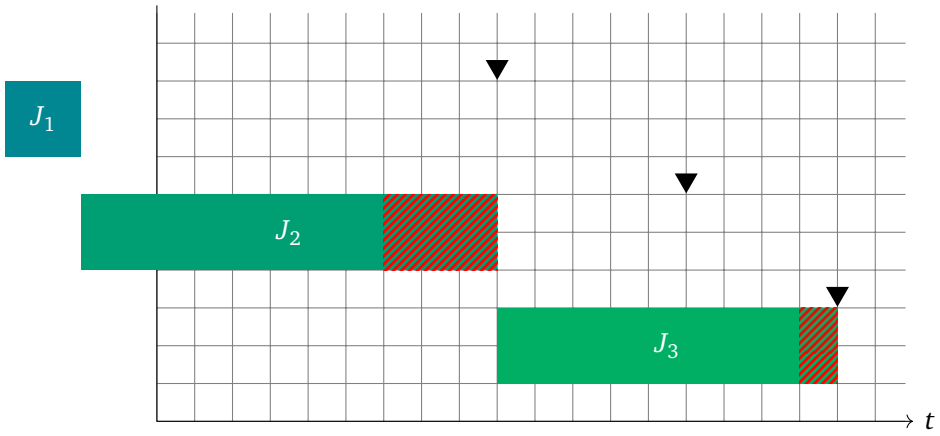
(a) An overloaded schedule



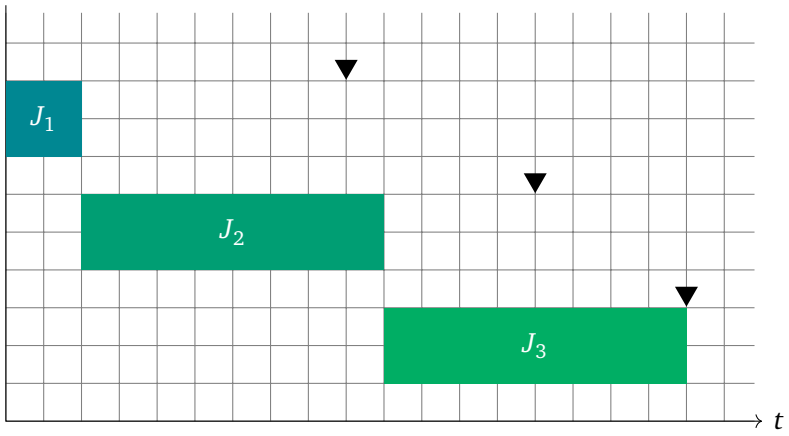
(b) The non-overloaded schedule after applying a laxity-based cutback to it

Figure 4.4: Application of a laxity-based cutback to the schedule

In Figure 4.4(a) and 4.4(b), a scenario is given that has jobs $J_1 = (e_1 : 2,\ d_1 : 2)$, $J_2 = (e_2 : 5,\ d_2 : 8)$, and $J_3 = (e_3 : 3,\ d_3 : 8)$. The respective laxity for each job is thus $\text{laxity}_1(0) = 0$, $\text{laxity}_2(0) = 3$ and $\text{laxity}_3(0) = 5$. Each job's cutback on run queue R_j can be calculated as $c_j \cdot \text{laxity}_i(0) / \sum_{J_k \in R_j} \text{laxity}_k(0)$. Thus, the cutback for job J_1 is 0, as it has no laxity. J_2 is cut back by $6/8$, resulting in a scheduled execution time of $\hat{e}_2 = 34/8$ and J_3 by $10/8$, yielding $\hat{e}_3 = 14/8$. After applying each cutback, the schedule looks like depicted in Figure 4.4(b).



(a) An overloaded schedule

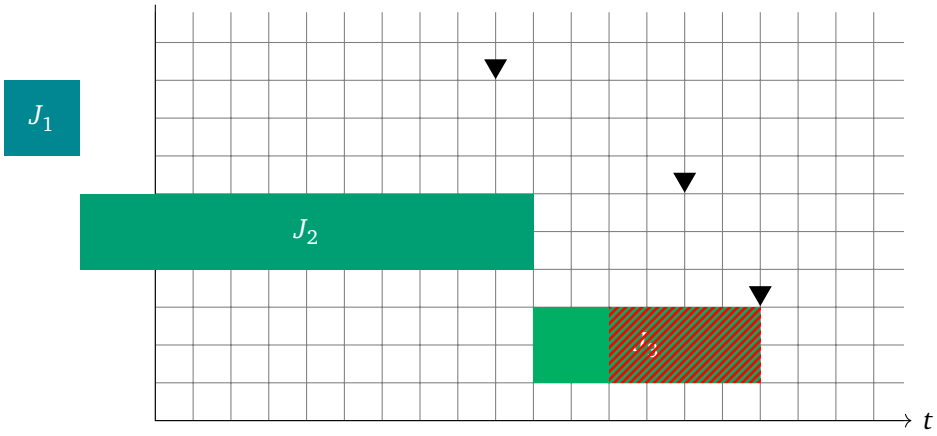


(b) The non-overloaded schedule after applying a cutback to fair share to it

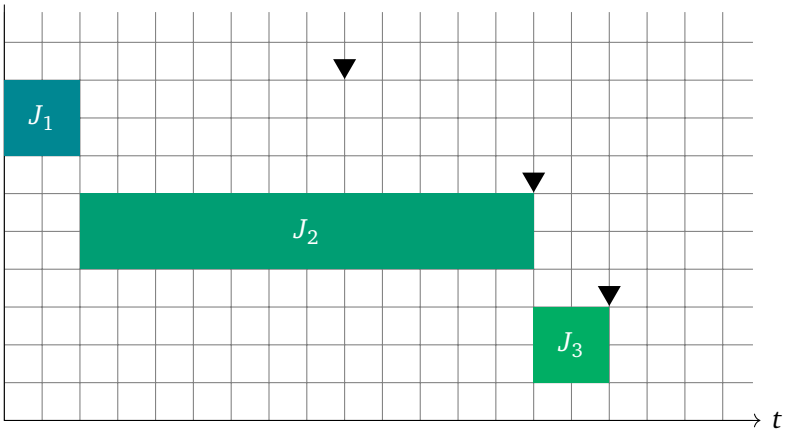
Figure 4.5: Application of a cutback to a fair share to the schedule

An example for the `OL_POL_FAIR` policy is given in Figure 4.5(a). Jobs $J_1 = (e_1 : 1,\ d_1 : 4.5)$, $J_2 = (e_2 : 5.5,\ d_2 : 8)$, and $J_3 = (e_3 : 4.5,\ d_3 : 9)$ exist in the system. The resulting fair share for three jobs and a latest deadline at time point 9 would be 3. As job J_1 already reserved less than the fair share, it can be subtracted from the available time and be left unmodified. Thus, for the remaining jobs J_2 and J_3, 8 time units are available, resulting in a fair share of 4. The required cutback for J_2 is therefore $^3/_2$ and for J_3 $^1/_2$. Note, that in the current implementation, all jobs scheduled execution times are set to 3 instead. For the given experiments, this does not pose a problem, as jobs actually execute for the amount of time they reserved. J_1 would thus only execute for one time unit and then donate the remaining execution time as slack to job J_2.

Cutback applied to the last jobs



(a) An overloaded schedule



(b) The non-overloaded schedule after applying a cutback to the last job

Figure 4.6: Application of a cutback to the last job

The application of the last policy, `OL_POL_DROP`, is depicted in Figures 4.6(a) and 4.6(b). Here jobs $J_1 = (e_1 : 1,\ d_1 : 4.5)$, $J_2 = (e_2 : 6,\ d_2 : 8)$, and $J_3 = (e_3 : 3,\ d_3 : 9)$ are known to the system. The required cutback of 2 is applied solely to the last job J_3, reducing its scheduled execution time to 1.

4.2.3 Configuration

The policies for overload detection and overload handling can be configured via sysctl parameters. Setting them can be done with the following variables:

- `kernel.sched_atlas_overload_detection` sets the used policy for overload detection as shown in Table 4.1. When using the ILF, the used window can be specified in microseconds by `kernel.sched_atlas_overload_ilf_window_us`. Though overload detection is used no more for overload handling, its type can still

```
struct atlas_load_stats {
  bool            has_ol;
  struct timeval  ol_slack;
  struct timeval  ol_ilf;
  struct timeval  ol_hlf;
  int             nr_jobs;
  struct timeval  rq_demand;
  struct timeval  rq_demand_all;
  struct timeval  rq_avail;
  bool            has_jobs;
  struct timeval  this_rexectime;
  struct timeval  this_exectime;
  int             this_class;
  struct timeval  this_rem;
};
```

Listing 4.2: The structure `atlas_load_stats`

be set and the result used from userspace with the `atlas_getload()` system call. The overload detection policy influences which policy is used to indicate overload with the `has_ol` field. The default is to use slack-based overload detection.

- `kernel.sched_atlas_overload_handling` determines, which policy is used to handle overload situations. Possible values are given in Table 4.2.

4.3 System Call

For communicating the current load situation from kernel to userspace, the system call `atlas_getload()` was implemented. This system call enables userland applications to get information on the schedule of ATLAS and about the currently executing job. It takes a pointer to a structure `atlas_load_stats` in userspace and fills it with the current values. The structure is depicted in Listing 4.2.

The fields of the structure represent the following:

- `has_ol`: A boolean value indicating, whether the system is in a state of overload. Whether the system is in overload is determined by the overload detection policy set by the sysctl variable `kernel.sched_atlas_overload_detection`.
- `ol_slack`: The amount of (possibly negative) slack of the first job.
- `ol_ilf`: The load returned by the ILF-based overload detection.
- `ol_hlf`: The load returned by the HLF-based overload detection.
- `nr_jobs`: The number of jobs currently enqueued in the atlas run queue of the calling process.
- `rq_demand`: The sum over all remaining execution times on the current atlas run queue.

```
template <typename Ret, typename... Args>
struct load_manager {
  load_manager(detection det = simple, operation op = alternative);

  void register_alternative(weightType weight,
                            packaged_task<Ret(Args...)> &&f);

  void register_iterative(packaged_task<Ret(checkFn, Args...)> &&f);

  Ret run(deadlineType dl, Args... args);
  ...
};
```

Listing 4.3: The class `load_manager`

- `rq_demand_all`: The sum over all remaining execution times and `sum_exec_runtimes` on the current run queue. Note that the `sum_exec_runtime` might be counted multiple times, if one thread executes multiple jobs. If jobs have separate threads executing them, this indicates how much computation time a job received in *all* scheduling bands of ATLAS and not just in the `ATLAS` band of ATLAS.
- `rq_avail`: This available time on the current run queue. This is equal to the time until the latest deadline on the current run queue.
- `has_jobs`: This boolean value indicates, whether the calling thread has jobs registered with ATLAS. The following values are all set to zero, it the thread does not have any jobs. Otherwise, they refer to the first job of the thread.
- `this_rexectime`: The received execution time in the `ATLAS` scheduling band.
- `this_exectime`: The submitted execution time.
- `this_class`: The scheduling band, this job is currently executed in. The value corresponds to the enumeration `atlas_classes` as defined in `kernel/sched/atlas.h` in the source tree of ATLAS.
- `this_rem`: The remaining execution time of this job. This also subtracts the time received while pre-rolling and if the job is in the `CFS recovery` band.

4.4 Userland

To avoid overload in the first place and to be able to react to overload situations from userland when they occur, a load manager was implemented. It makes use of the `atlas_getload()` system call presented in Section 4.3. The templated class `load_manager<Ret, Args...>` is defined as a header-only implementation. The interface to use the load manager is given in Listing 4.3 and will be described next.

As the load manager returns the result of the calculation when `run()` finishes, it needs to be instantiated with the correct template arguments. Those arguments are the return

type `Ret` and the argument types `Args...` of the functions that will be registered with the load manager. After it has been instantiated, callable objects such as function pointers, C++ lambdas, or block from the BlocksRuntime, can be registered with it. One of the registered callables will be executed when calling the `run()` method.

4.4.1 Modes of Operation

The load manager knows two different modes of operation, one of which has to be specified when instantiating it. The possible values are:

- `lm::operation::alternative` specifies that the load manager should handle different alternative functions. All these functions have to have the same signature. The functions are registered using the `register_alternative` method. In addition to the function object, this method takes a weight parameter, that is currently used as the predicted execution time. When calling the `run()` method, the function most suitable for the current load situation is chosen and executed as an ATLAS job.
- `lm::operation::iterative` allows the user to register one function that can iteratively refine its result via `register_iterative`. This function has to take a function of type `lm::load_manager::checkFn` as its first argument. The `checkFn` is provided by the load manager, takes no arguments, and returns a `bool` that indicates, whether the system is currently overloaded and the iterative refinement function should cease iteration.

Internally the function objects are stored as `packaged_task<Ret(Args...)>` and `packaged_task<Ret(checkFn, Args...)>` respectively.

4.4.2 Execution Times and Deadlines

When executing `run()`, a deadline has to be specified that is used as the absolute deadline for the job that is submitted to ATLAS. For functional alternatives, currently the `weightType` is a `std::chrono::duration` that is used as the job's submitted execution time. For iterative refinement functions, 97.5 % of the available time until the deadline is used as the submitted execution time.

4.4.3 Example—Functional Alternatives

Listing 4.4 shows how different callables can be registered with the load manager. First, in line 5, an instance of `load_manager` is instantiated that handles functional objects that have the return type `int` and take a reference to a `std::string` as their only argument.

In line 6, the function `fn` that was declared in line 1 is registered with a weight of 2 ms. Internally, the passed function pointer is encapsulated within a `packaged_task<int(std::string&)>` and moved to a list. As a packaged task can hold different types

```
1  int fn(std::string &s);
2
3  int main() {
4    std::string s("Razupaltuff");
5    ...
6    load_manager<int, std::string&> lm(simple, alternative);
7    lm.register_alternative(2ms, fn);
8    lm.register_alternative(3ms, [](std::string &s){ ... });
9    lm.register_alternative(4ms, ^(std::string &s){ ... });
10   int ret = lm.run(10ms, s);
11   ...
12 }
```

Listing 4.4: An example of using functional alternatives with the load manager

of callable objects, it is possible to register lambdas with the load manager. In line 7, a lambda is passed as an r-value reference to the function with a weight of 3 ms. Similarly, in line 8, a BlocksRuntime block registers with the load manager using the weight 4 ms.

After registering the three callable objects with the load manager, the `run` method is called with an absolute deadline of 10 ms. This call causes the load manager to look up the current load situation using the `atlas_getload` system call and decide which callable fits best. The load manager then submits a job to ATLAS using the provided deadline and the execution time of the callable it selected. When the job has been submitted, the load manager executes it and waits for it to finish. After execution has stopped, the value returned by the callable object is returned as `run()`'s return value.

4.4.4 Example—Iterative Refinement

The skeleton of an iterative refinement function can be seen in Listing 4.5. Lines 1 to 8 define a function that takes two arguments, one of type `checkFn` and one of type `int`. The function then initializes the return value with a value of 42 in line 2. Thereafter, the function uses a while loop to iteratively refine the return value `ret`. The while loop stops when either `stopCond` or a call to the `checkFn` evaluate to `true`: either the function decides so itself via the `stopCond`; or, the load manager indicates with the `checkFn()`, that the system is overloaded or near to overload and the function should stop execution. After the loop has stopped, the function has to return the result as soon as possible to avoid further delay.

In the `main` function first the load manager is instantiated. The constructor sets the load managers mode of operation to handle iterative refinement functions. Next, in line 13, the iterative refinement function `it_fn`, that was just described, is registered with the load manager. In this case, no `weightType` is needed. Either the function will be executed until it terminates regularly or the system is overloaded and the function needs to stop to reduce the load on the system. In line 14, the load manager is instructed to execute the function with an absolute deadline of 42 ms and the argument 23. Like in the mode of operation for functional alternatives, an ATLAS job will be submitted and then executed by the load manager. The ATLAS job will have the provided deadline

```
double it_fn(lm::checkFn cfn, int arg) {
  double ret = 42;
  ...
  while (!stopCond && !cfn()) {
    ... refine ret ...
  }
  return ret;
}

int main() {
  ...
  load_manager<double, int> lm(simple, iterative);
  lm.register_iterative(it_fn);
  double res = lm.run(42ms, 23);
  ...
}
```

Listing 4.5: An example of using iterative refinement functions with the load manager

and an execution time of 97.5 % of the available time until its deadline to account for overhead before and after the loop.

5 Evaluation

To evaluate the modifications made to the ATLAS kernel, this chapter presents the results of the experiments that were run. First, to achieve reproducibility, Section 5.1 will give an overview of how the kernel part was developed and how experiments where run. Then, in Section 5.2, the experiments and results for the load manager will be shown. Last, Section 5.3 presents the evaluation of the in-kernel overload handling.

5.1 Setup

To allow the results in Section 5.2.1 and 5.2.2 to be reproduced, this section will give an overview of the used hardware and software components. It will include the setup while developing as well as the setup used to run experiments.

5.1.1 Development

Development was done using the QEMU emulator. QEMU offers the possibility to boot a Linux kernel by appending `-kernel` to the QEMU command line. Furthermore it is possible to specify the kernel command line with the `-append` option. This was used to instruct the kernel to output to the serial terminal, so the output can be directly output by the terminal emulator used.

The complete command line used for starting QEMU can be seen in Listing 5.1. `BZIMAGE` holds the path of the ATLAS kernel. Also, as this work only focuses on unicore systems, QEMU was instructed to only emulate a single CPU. The `-nographics` switch disables graphical output as only text output to the serial console was used. Last, `-enable-kvm` was set to make use of the host system's Kernel-based Virtual Machine (KVM)-capabilities and speed up emulation.

```
1 qemu-system-x86_64 \
2   -nographic \
3   -enable-kvm \
4   -s \
5   -m 1G \
6   -smp 1 \
7   -kernel "${BZIMAGE}" \
8   -append "console=ttyS0"
```

Listing 5.1: QEMU invocation

```
#!/bin/busybox sh

export PATH='/bin:/sbin:/code'

dmesg -n1

/bin/busybox --install -s /bin

# Mount the needed filesystems.
mount -nt proc proc /proc
mount -nt sysfs sysfs /sys
mount -nt debugfs debugfs /sys/kernel/debug
mount -nt devtmpfs devtmpfs /dev

# we are connecting via serial console, so redirect all I/O
exec 1>/dev/ttyS0 2>/dev/ttyS0 </dev/ttyS0

# set up atlas' debugfs
/bin/debug overload ol_detection ol_handling ol_stats

dmesg -c >/dev/null

exec /bin/init
```

Listing 5.2: `/init` script in the initramfs

For development, as well as for testing, the kernel was build using an *initramfs* built into the kernel. This can be achieved by setting the `CONFIG_INITRAMFS_SOURCE` kernel option to a path relative to the source directory, that holds the initramfs to build into the kernel.

This offers two advantages. First, it removes the need for a separate root file system. As the initramfs is packed every time the kernel is build, there is also no risk of persistently modifying data on the root file system which may influence the behaviour of the system. And second, this removes the possibility of disk I/O influencing the measurements[1]. As the initramfs is part of the kernel, it is loaded together with the kernel into memory at startup.

Instead of a fully-fledged init system like System V Init or systemd, I used the init system provided by busybox. Busybox offers many of the utilities, GNU fileutils, shellutils and others usually installed on Linux systems provide while itself being only a single binary that may be statically linked.

I placed the compiled binary for busybox as well as the binaries to run experiments in the initramfs to be packed and included in the Linux kernel. After the system booted, they were executed from the shell, that busybox init presents to the user.

Listing 5.2 shows the init script used to set up the system, that I used for development as well as the experiments.

[1] While caches also influence the measured result, those influences are orders of magnitude smaller than latencies introduced by disk read/seek.

First, the `PATH` variable is set to where the binaries are located. Line 6 causes only emergency messages to automatically appear on the console. Then, busybox installs symlinks for the applications it includes into the `/bin` directory. As some programs require `/proc`, `/sys`, and `/dev` to be mounted, those are set up in lines 10 to 13 along with `/sys/kernel/debug` that is used to control ATLAS' debug output. When `/dev` is ready, all input as well as output to `stdout` and `stderr` are redirected to the serial console `/dev/ttyS0`. In line 19, a script is called, that was written to set up ATLAS' debugfs to only the values given as arguments. In the following, `dmesg -c >/dev/null` is used to clear the kernel message ringbuffer and not display it. Finally—in line 23—control is handed to busybox' init system that starts a shell.

5.1.2 Experiments

Experiments were performed on an Intel Core i7 CPU that is clocked at 2.67 GHz. I turned off frequency scaling and Hyper-Threading in the BIOS. The machine has 4 GiB of DDR3 random access memeory (RAM) running with a clock frequency of 1067 MHz.

To start experiments and for debugging purposes, I connected to this machine via its serial port. As can be seen in Section 5.1.1, everything needed is already included in the kernel image generated in the build process. Thus, I only needed to provide this kernel image to the machine[2] by Preboot Execution Environment (PXE).

PXE is a feature of certain basic input/output systems (BIOS). It starts by acquiring an Internet Protocol (IP) address via the Dynamic Host Configuration Protocol (DHCP). The DHCP server also provides information, about where the host can find the kernel image to load and how it is called. Then, the kernel image is retrieved from this address via the File Transfer Protocoll (FTP). For this thesis, I used dnsmasq as a DHCP-server and tftp-hpa as the FTP-server that provides the kernel image that was built as described in Section 5.1.1. Furthermore, I specified the kernel command line options `console=ttyS0,115200 maxcpus=1` to instruct Linux to output to the serial console and only use a single CPU.

5.2 Overload Avoidance

To give userspace applications the ability to avoid overload in the first place, I implement a userspace component, the load manager. The next sections will take a closer look at both of the load manager's modes of operation. Section 5.2.1 will look at the mode for functional alternatives, while Section 5.2.2 will shine light on how the iterative refinement mode performed.

[2] Actually, the hard drive was even physically disconnected.

5.2.1 Functional Alternatives

This section will first describe the setup of the experiment used for functional alternatives and then the results, this experiment yielded.

Experiment Setup I used the bodytrack benchmark from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [BL11] benchmarking suite to evaluate functional alternatives. JouleGuard [Hof15] uses *approximate computing* to trade accuracy for battery lifetime. One of their experiments also leverages the bodytrack benchmark. They use PowerDial [HSC+11], that uses "knobs" in applications to change their accuracy. For bodytrack, those "knobs" are the number of particles and the number of annealing layers used by its algorithm. To emulate PowerDial's behavior, I renamed the `main()`-function of the bodytrack benchmark to `bodytrack_main()`, added a header that declares this function and built bodytrack as a static library. Then, I added a new executable, that links against the bodytrack library, constructs an argument vector, and calls the `bodytrack_main()`-function with this argument vector. As seen in Figure 5.1, the bodytrack benchmark scales linearly to the used number of layers and particles respectively.

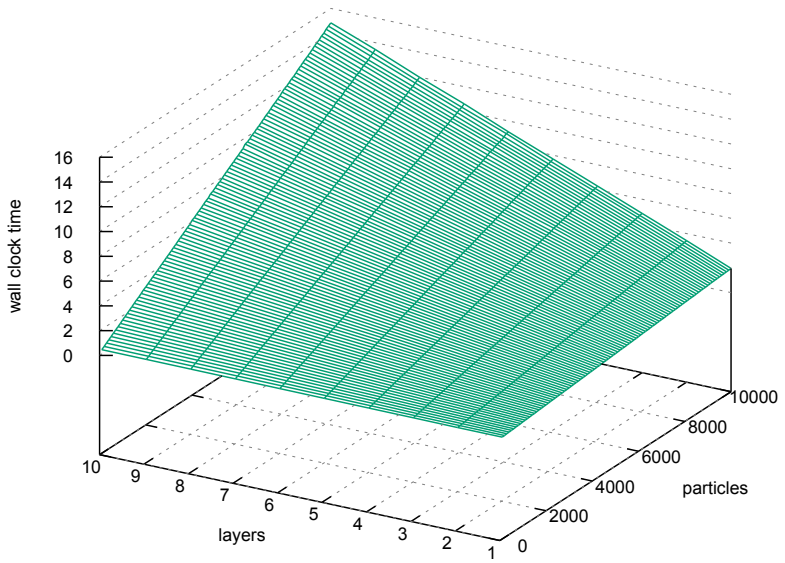


Figure 5.1: Influence of the parameters on the wall clock time of bodytrack

To test the load managers functional alternative mode, a background load is required to see how an adaptive application fares with different levels of background load. Therefore, the experiment starts a single periodic task. The period is set to 100 ms and the utilization it generates varies. The utilization had values in the range from 0.1 to 0.9 in

#	Layers	Particles	Execution time
0	-	-	<0.01 s
1	1	100	0.21 s
2	5	500	0.56 s
3	10	500	0.96 s
4	3	2300	1.29 s
5	6	1600	1.70 s
6	6	2000	2.04 s
7	2	7100	2.58 s

Table 5.1: Parameters and execution times used with the functional alternatives experiment

steps of 0.1 that determine the execution time of jobs of this task as

$$e_i = U_p \cdot \text{period}$$

Additional to the load generated by the task, a load manager is instantiated, and the eight different parameter combinations in Table 5.1 for the bodytrack benchmark registered with it.

As the latest relative deadline in the job set is be 2.5, I chose those alternatives with execution times up to 2.5 s. The alternatives used in the experiment are shown in Table 5.1. The experiment first checks, whether the file `/etc/bodytrack_params` exists. If not, the different alternatives are measured and the results written to this file. If the file exists, the values stored there are used as the weight parameter[3] for the load manager. The setup of the load manager and measuring/loading of bodytrack's execution times is done prior to starting the task. After the task is started, the load manager's `run()` function is called with the latest deadline used by the task, causing it to select an alternative and run the bodytrack benchmark. The measured execution times for the different alternatives, that the experiment uses as weight parameters for the load manager are also listed in Table 5.1.

For this setup, I measured the tardiness of each job of the task as well as for the load manager. Also, I recorded the functional alternative used by the load manager.

Results In Figures 5.2(a) to 5.2(d) the dots show the tardiness of the periodic set of background jobs that is scheduled together with a functional alternative chosen by the load manager. The vertical red line represents the shared deadline of the functional alternative and the last background job. The triangle represents the load manager that always finishes after the other jobs if it chose an alternative other than the shortest one. The shortest option is just a call to a function that immediately returns, which can be executed in the slacktime present in the schedule. As the load manager shares its deadline with the last job, which job finishes last depends on the submission order. Subscribed under the triangle and shown in Table 5.2 is the execution time of the chosen

[3]The weight parameter is currently used as the submitted execution time for the chosen alternative. In future versions it might be used to train ATLAS' predictor.

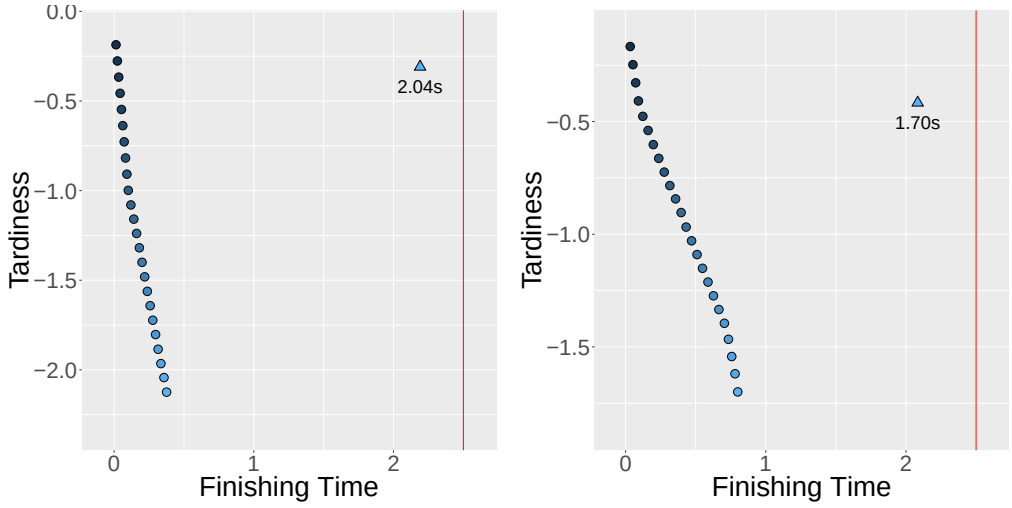
U_p	Alternative	Available Time	Execution Time
0.1	#6	2.25 s	2.04 s
0.2	#5	2.00 s	1.70 s
0.3	#4	1.75 s	1.29 s
0.4	#4	1.50 s	1.29 s
0.5	#3	1.25 s	0.96 s
0.6	#2	1.00 s	0.56 s
0.7	#2	0.75 s	0.56 s
0.8	#1	0.50 s	0.21 s
0.9	#0	0.25 s	<0.01 s

Table 5.2: Alternatives chosen by the load manager with different utilizations

alternative used by the load manager. As can be seen in Figure 5.2(a), with a utilization of $U_p = 0.1$, the load manager has 90 % of the time until its own and the deadline of the latest job at time 2.5 s. As seen in Table 5.2, the load manager thus chooses an alternative with an execution time of approximately 2.04 s and executes it. For a utilization of $U_p = 0.2$, the available time is 2.0 s and the load manager chooses the alternative with an execution time of 1.70 s. The same holds for the utilization 0.5. For a utilization of 0.9, the load manager chose to execute the shortest alternative that just calls a function that returns immediately.

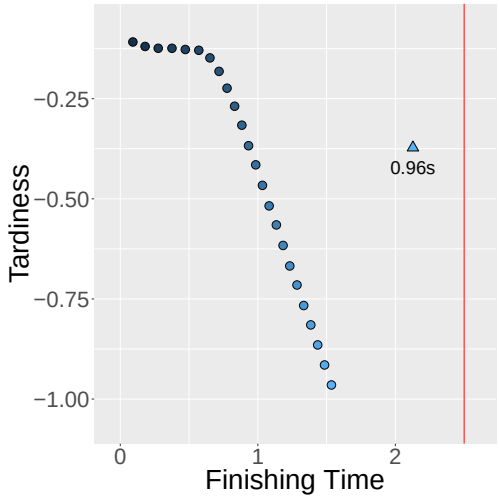
These exemplary runs show that the load manager does not cause jobs that are already scheduled to miss their deadlines. Also, it achieves to choose an alternative, that fits the schedule, so that itself does not miss its deadline. The only influence on the background jobs is that they finish at a lower rate than without the adaptive application, but still prior to their deadline.

The current implementation uses the load calculated by the Horizon Load Function (HLF). Because the last deadline of the job set and the load manager coincide, the HLF is accurate. If the deadline of the load manager were later than the latest deadline, the HLF would overestimate the load and could cause the load manager to not chose the optimal alternative. Another possibility—though a suboptimal one—would have been to set the interval used by the ILF to the time until the load manager's deadline via sysctl. A better option would be to extend the `atlas_getload` system call to take a second parameter that represents the timespan, the calling process is interested in. If this parameter is `NULL`, as a default value the latest deadline in the schedule could be used. As the levels for the load managers alternatives were chosen too coarsely, its execution in some of the examples ends a long time prior to the deadline. A better result would be achieved, if there were more alternatives and if they were more evenly spread.

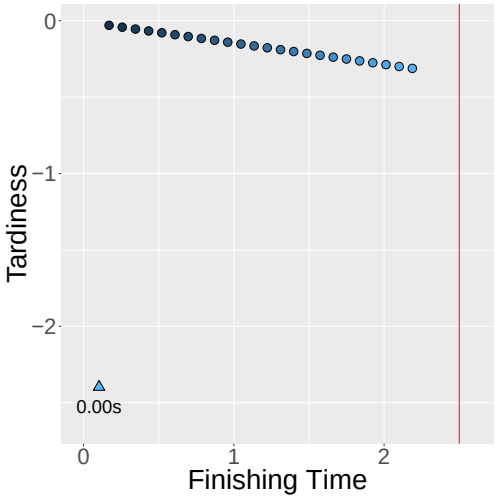


(a) $U_p = 0.1$

(b) $U_p = 0.2$



(c) $U_p = 0.5$



(d) $U_p = 0.9$

Figure 5.2: The load manager using functional alternatives

5.2.2 Iterative Refinement Functions

Next to functional alternatives, a method to avoid overload provided by the load manager are iterative refinement functions. In this section I will first describe the performed experiment and then evaluate the results this experiment produced.

Experiment Setup To evaluate the iterative refinement mode of the load manager, I adapted an algorithm from one of the examples provided with Mitch Richling's GNU Multiple Precision Arithmetic Library (GMP) examples [Ric99] to iteratively calculate π. While this algorithm is not very efficient, a single iteration does not take very long, so the algorithm can stop its refinement and yield a result with low overhead when determined by the load manager.

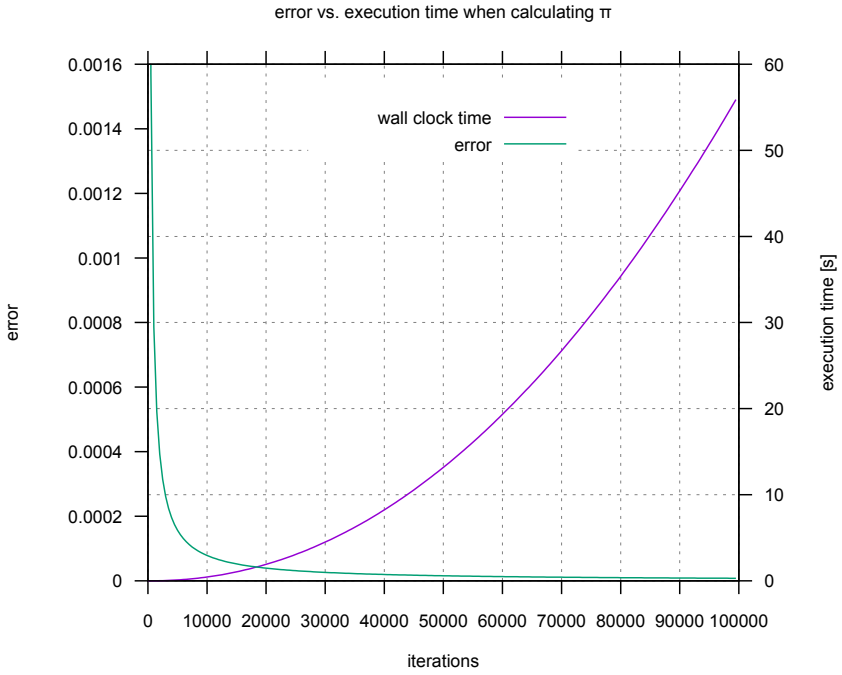


Figure 5.3: Error and wall clock time of the iterative π calculation

In Figure 5.3, the x-axis shows the number of iterations performed. The left y-axis along with the purple graph shows the wall clock time, a run with this number of iterations took. The green graph—with its axis on the right side—represents the error to the real value of π. As can be seen, the error asymptotically approaches zero the more iterations are calculated while the wall clock time rises exponentially. This is due to the arithmetic operations of GMP taking longer, the more precision is required.

This experiment also needs a competing load. I chose a periodic task that submits and executes 25 jobs. The period is again set to 100 ms. After the task is created, the load manager is started in its own thread. Then, the function for iteratively calculating π is registered with it. Finally, the `run()` method of the load manager is called with a

relative deadline of 2.5.

In this experiment, I measured the tardiness of jobs of the task and of the load manager. Furthermore, I recorded the number of iterations performed by the iterative refinement function.

Results The experiment for using iterative refinement functions with the load manager, shown in Figures 5.4(a) to 5.4(d), tested how the load manager performs in front of a background load generated by one ATLAS task. The plots depict the tardiness of the jobs of the background task and of the iterative refinement function. The jobs of the periodic task are depicted by the dots while the load manager is again represented by the triangle. The number of performed iterations is written below the load manager's triangle. The vertical red line represents the deadline of the last back job and the load manager.

For a utilization of $U_p = 0.1$, the iterative refinement function can use $0.9 \cdot 2.5\,\text{s} = 2.25\,\text{s}$. In this time, it is able to perform 21 557 iterations. When the utilization is increased, the number of iterations performed decreases and the rate at which the background jobs finish drops. In the heavily loaded system shown in Figure 5.4(d), the iterative refinement function can only achieve 8355 iterations. The number of iterations performed does not decrease linearly, as the function for iteratively calculating π scales exponentially (cf. Figure 5.3).

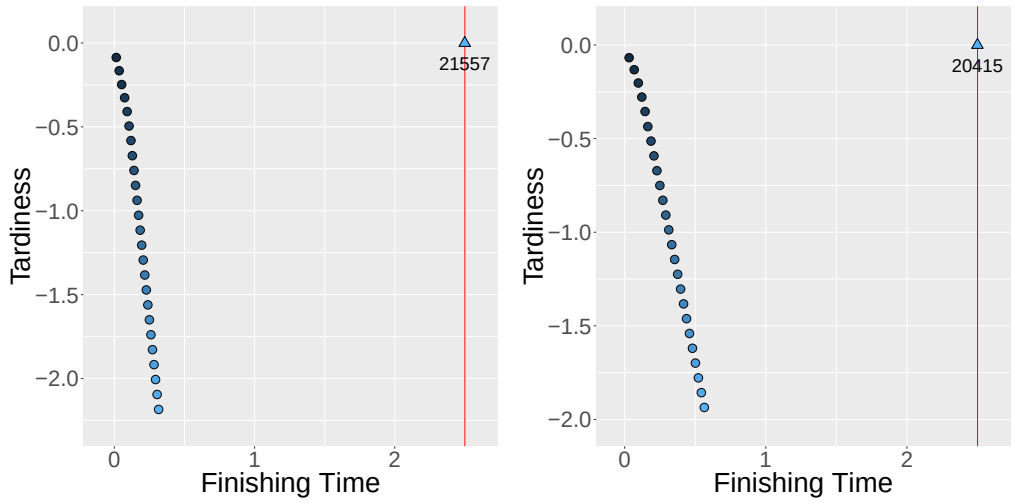
This experiment shows, that like with the load managers mode of operation for functional alternatives, the background jobs are only affected in the rate at which they finish. Calculation of the iterative refinement function does not interfere with the execution of the background jobs. Also the load manager finished just before its own deadline, thus maximizing the amount of time available for execution of its function.

5.3 Overload Handling

As overload cannot always be avoided, I extended the kernel with overload handling facilities. For evaluating the in-kernel overload handling, an experiment was written that is described in Section 5.3.1. Thereafter, Section 5.3.2 will present the results of this experiment.

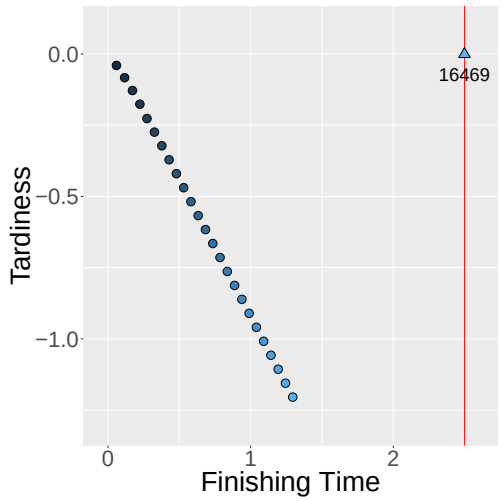
5.3.1 Setup

The experiment for testing overload handling consists of one or two periodic tasks. Each task starts 25 jobs with a period of 100 ms. The task is implemented as a thread that submits all jobs in the beginning and then executes them in the order ATLAS tells it to via the `atlas_next()` system call. The jobs in each task have monotonically increasing deadlines that are one period apart, so the order in which jobs of a task are executed is deterministic. Over this task set I varied the generated processor utilization U_p at levels that are shown in Table 5.3.

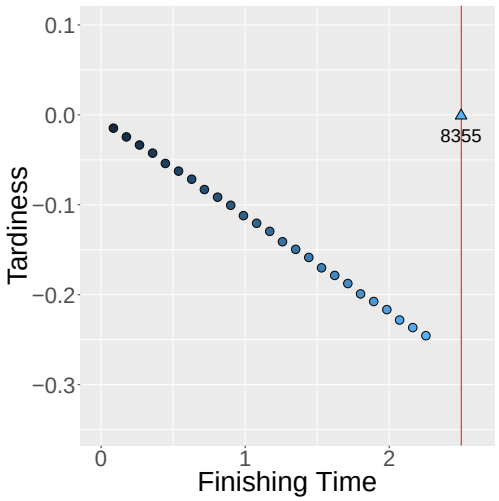


(a) $U_p = 0.1$

(b) $U_p = 0.2$



(c) $U_p = 0.5$



(d) $U_p = 0.9$

Figure 5.4: The load manager using an iterative refinement function

U_p\ratio	0	0.05	0.1	0.25	0.4	0.5
0.95	0/95	4.75/90.25	9.5/85.5	23.75/71.25	38/57	47.5/47.5
0.99	0/99	4.95/94.05	9.9/89.1	24.75/74.25	39.6/59.4	49.5/49.5
1.0	0/100	5/95	10/90	25/75	40/60	50/50
1.05	0/105	5.25/99.75	10.5/94.5	26.25/78.75	42/63	52.5/52.5
1.1	0/110	5.5/104.5	11/99	27.5/82.5	44/66	55/55
1.2	0/120	6/114	12/108	30/90	48/72	60/60
1.5	0/150	7.5/142.5	15/135	37.5/112.5	60/90	75/75
2.0	0/200	10/190	20/180	50/150	80/120	100/100

Table 5.3: e_1/e_2 in ms for each configuration

To test how jobs with different execution times affect the system, another dimension of the experiment is the ratio of execution times of the jobs to each other. Among the two tasks, the total utilization is split by the ratio of the execution times of the jobs. Thus, the execution times for jobs of the first task is calculated by

$$e_1 = U_p \cdot \text{ratio} \cdot \text{period}$$

and as

$$e_2 = U_p \cdot (1 - \text{ratio}) \cdot \text{period}$$

for jobs of the second task. The tested levels for the ratio are also shown in Table 5.3. So, for example, a period of 100 ms, a total utilization of 1.5, and a ratio of 0.25 result in an execution time of 37.5 ms for jobs $J_{1,i}$ of the first task and 112.5 ms for jobs $J_{2,i}$ of the second task.

For all those combinations, the experiment was run 25 times for each overload handling policy. In each run, I measured the tardiness of each job.

5.3.2 Results

This section presents the results that the experiment for overload handling yielded. First, I will show the experiment for a single task and then the experiment for two tasks with a different ratio of execution times in an underloaded system. Finally, I show how two tasks with a ratio of execution times of $e_1/U_p = 0.25$ behave when they are scheduled by the different policies implemented in the ATLAS kernel.

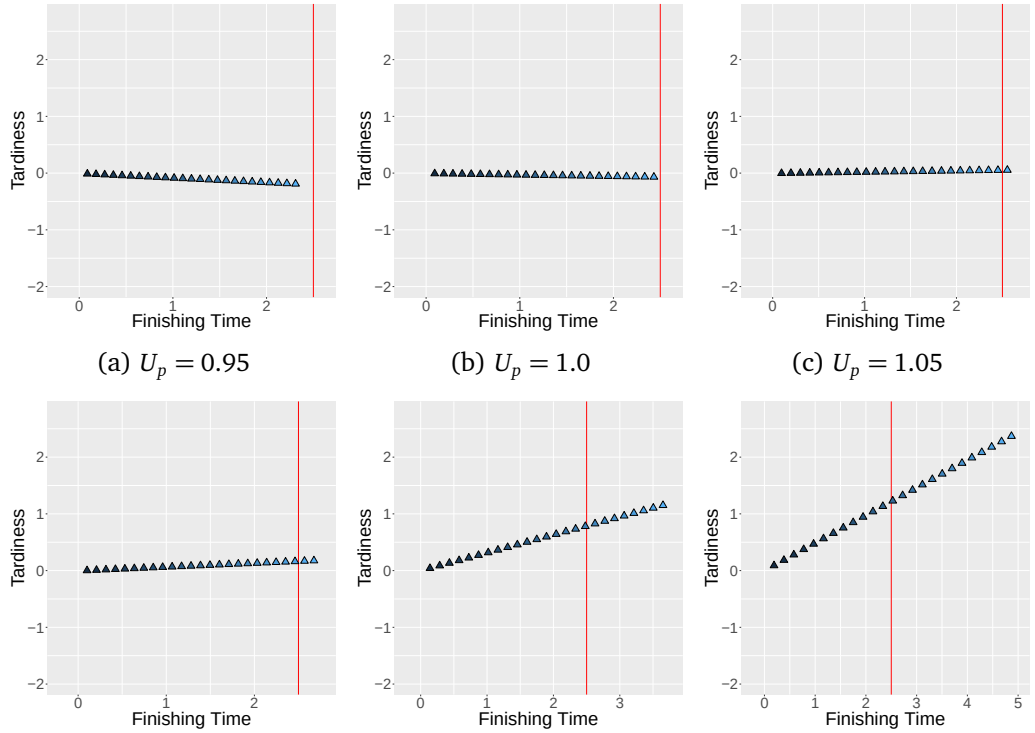
A Single Task

Figure 5.5 shows the tardiness of 25 jobs of a single task with a period of 100 ms. Beginning with Figure 5.5(a) to Figure 5.5(f), the system's utilization is increased from 0.95 to 2.0. In the first Figure, the tardiness of the jobs decreases over time. As the system has a utilization of $U_p = 0.95$ and is therefore underloaded, slack accumulates. Jobs can use this slacktime to start earlier and finish before their deadline. Even in Figure 5.5(b), the tardiness still decreases over time. This is due to the jobs only working

about 97.5 % of their submitted execution time to account for overhead. As the jobs' tardiness decreases over time, this margin was obviously chosen too broad.

Starting from Figure 5.5(c), the system is definitively overloaded, as even though jobs work only for 97.5 % of their submitted execution time, in this example they at least require $105\,\text{ms} \cdot 97.5\,\% \approx 102\,\text{ms}$ of execution time to finish.

For a utilization of $U_p = 1.05$, the total demand of all jobs is $25 \cdot 105\,\text{ms} = 2625\,\text{ms}$. The available time until the latest deadline is 2500 ms. This means, that the schedule is overloaded by 125 ms. As explained in Section 2.4.1, the schedule is constructed from the latest deadline in reverse order. In case of an overloaded system, this causes jobs to be scheduled in the past. Therefore, jobs at the beginning of the schedule have their scheduled deadline in the past. The first job would, for example, be scheduled from $-125\,\text{ms}$ to $-20\,\text{ms}$. As the scheduled deadline is in the past, ATLAS moves this job to `EDF Recovery`. The first job would actually be executed on the time of the second job, which is still in the `ATLAS` band. This continues until the point in time, at which the current time passes the scheduled deadline of the currently executing job. After that, the remaining jobs are moved to `CFS` and executed there.



(a) $U_p = 0.95$ (b) $U_p = 1.0$ (c) $U_p = 1.05$

(d) $U_p = 1.1$ (e) $U_p = 1.5$ (f) $U_p = 2.0$

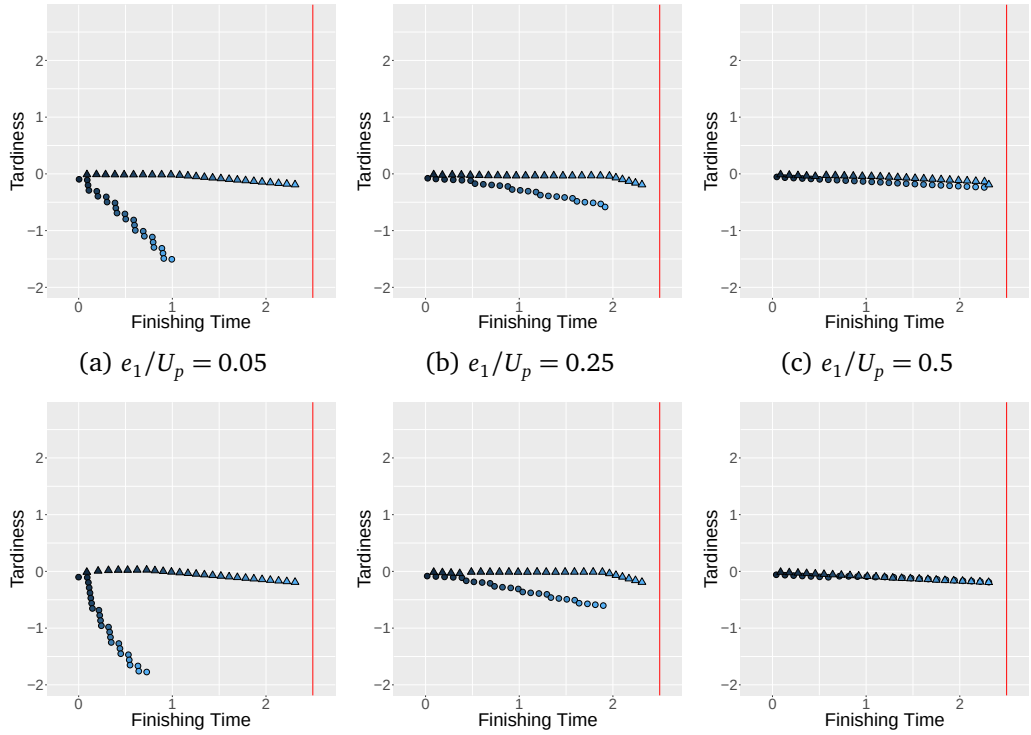
Figure 5.5: A system with a single task in different load situations

Two Tasks in an Underloaded System

The Figures 5.6(a) to 5.6(f) show a system with a load of $U_p = 0.95$. The system schedules two tasks that have a different ratio of execution times. On the left, the ratio

of task 1 is 0.05, in the middle 0.25, and on the right both tasks have the same execution time. The top row shows the system with no overload handling, the bottom row when using the fair policy. All policies, except the fair policy, check whether overload exists before modifying the schedule. If not, they just return. Therefore, the top row is also representative for all other policies.

For no overload handling, long jobs finish at about their period. The short jobs finish at a higher rate, as they can actually finish in the available slacktime generated by the schedule. In each period, the generated slack is 5 ms and short jobs in Figure 5.6(a) are only 4.75 ms, one to two jobs can finish in slack. After all jobs of task 1 have finished, the rate for jobs of task 2 increases until all have finished. With a ratio between execution times closer to one, this effect decreases, as seen in Figure 5.6(b) and 5.6(c), and the tardiness of jobs of each task equalizes.



(a) $e_1/U_p = 0.05$ (b) $e_1/U_p = 0.25$ (c) $e_1/U_p = 0.5$

(d) $e_1/U_p = 0.05$ (e) $e_1/U_p = 0.25$ (f) $e_1/U_p = 0.5$

Figure 5.6: A system with two tasks using no policy (top row) and the policy `OL_POL_FAIR` (bottom row) with a different ratio of execution times

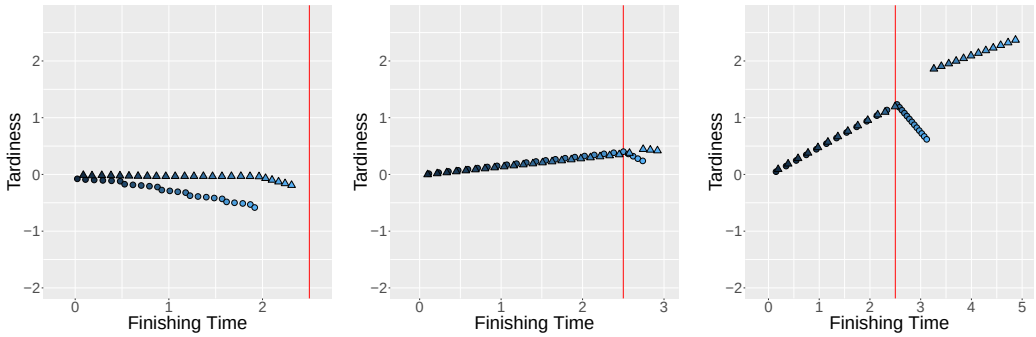
The only exception is the policy `OL_POL_FAIR`, which sets the scheduled execution times of jobs to their fair share regardless of the utilization of the system. As the jobs of task 1 only require 4.75 ms, slack of 45.25 ms is generated. Jobs of task 2 require an execution time of 90.25 ms but only receive 50 ms, they cannot finish within the scheduled execution time and are moved to `EDF Recovery`. There, they gain the slack generated by the jobs of task 1. So, even with the job of task 2 in `EDF Recovery`, two to three jobs of task 1 can finish per period. After the jobs of task 1 have finished, the rate at which jobs of task 2 finish increases. Thus the tardiness of those jobs decreases.

Two Tasks in an Overloaded System

As seen in the previous sections, most policies do not change the schedule in an underloaded system. For an overloaded system, this section shows the effects that occur with the different policies for three utilization levels. For all experiments in this section, the ratio of execution times is 0.25.

No overload handling For scheduling two tasks with a skew ($e_1/U_p = 0.25$) of execution times and a total utilization $U_p = 1.2$ without overload management, Figure 5.7(b) shows the tardiness of each job. Jobs of the task 1 have an execution time of 30 ms, job of task 2 execute for 90 ms.

The schedule is constructed by ATLAS from the latest deadline. As the system is overloaded, some jobs are scheduled at a point in the past. Those jobs are directly moved to `EDF Recovery`.

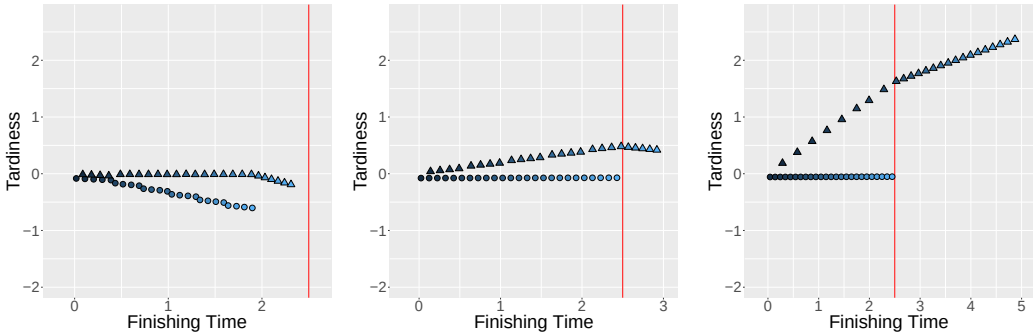


(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

Figure 5.7: Policy: `OL_POL_NONE`

The jobs of each task alternatingly receive execution time in the `ATLAS` band. The jobs start execution on the first job that is still in the `ATLAS` band. When a job cannot finish in time, it executes the remainder of its execution time on its successors reserved execution time when it is scheduled. After the last deadline at time 2.5 s has passed, the remaining jobs are moved to `CFS` and finish execution there. As tasks in `CFS` receive time independent of their deadlines and submitted execution times, the shorter jobs are able to finish in quick succession while the longer jobs finish after them. This can both be seen in Figures 5.7(b) and 5.7(c) for a utilization of 1.2 and 2.0 after the last deadline at time 2.5 s.

Policy: `OL_POL_FAIR` When scheduling with the fair policy, both tasks receive a fair share of 50 ms in the `ATLAS` band. This produces a schedule that alternatingly assigns 50 ms to jobs of each task. The modified schedule is not overloaded, but jobs of task 2 cannot finish, as they require an execution time of 90 ms for a total utilization of 1.2 and of 150 ms for $U_p = 2.0$. The short jobs can always finish on their timeslice, as they require at most 50 ms for a utilization of 2.0.



(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

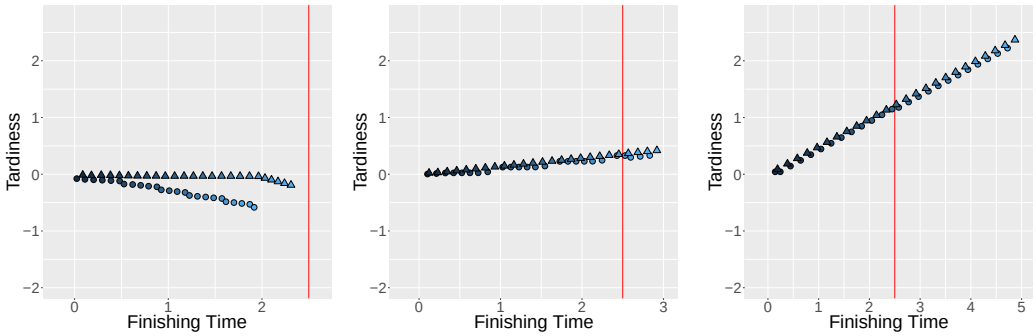
Figure 5.8: Policy: `OL_POL_FAIR`

This can be seen in Figure 5.8(b) where one job of task 1 finishes per period. As the short job does not require the 50 ms assigned to it, slack is generated. The job of the other task executes the full 50 ms, but does not finish. Therefore, it is moved to `EDF Recovery`, where it can use the 20 ms of slack generated by the job of task 1. In total, the job of task 2 can execute for 70 ms, which is not sufficient for it to finish. Thus, the completion of jobs of task 2 is not monotonic, which can be seen as gaps, for example shortly after time 1 s.

After the last deadline, all jobs of task 1 have finished, while jobs the remaining jobs of task 2 are moved to `CFS`. There they finish at a higher rate, as the competing load from jobs of task 1.

Figure 5.8(c) shows the situation for a higher utilization, causing jobs of task 2 to initially finish at a lower rate than with $U_p = 1.2$.

Policy: `OL_POL_FIXED` The fixed policy cuts back each job's execution time reservation in the `ATLAS` scheduling band by the same amount of time. $U_p = 1.2$ requires the overload handling to cut back 20 ms per period which is distributed among the jobs in every period by 10 ms each. Every short job is cut back from 30 ms to 20 ms and every long job from 90 ms to 80 ms.

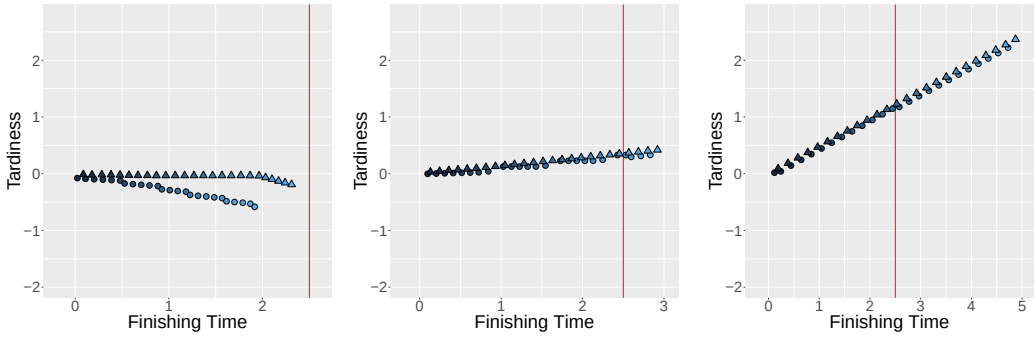


(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

Figure 5.9: Policy: `OL_POL_FIXED`

In Figure 5.9(b), jobs execute on their scheduled execution time. As the scheduled execution is not sufficient for jobs of either task to finish, they continue execution on the next job's time. This continues until the latest deadline, when all remaining jobs are moved to CFS, where they are scheduled independent from their deadline and execution time. The steps that can be seen in Figure 5.9(b) can currently not be explained and require more research.

Policy: `OL_POL_PROP` The policy `OL_POL_PROP` cuts back jobs proportionally to their reserved execution times. Thus, with a ratio of execution times of 0.25 the short jobs get cut back from 30 ms to 25 ms and long jobs from 90 ms to 75 ms.



(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

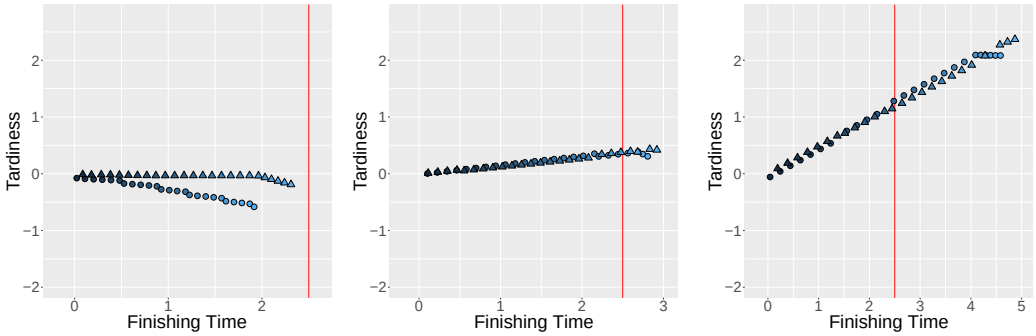
Figure 5.10: Policy: `OL_POL_PROP`

Like with the policy `OL_POL_FIXED`, this causes all jobs to be scheduled on their successors time. When the scheduled deadline of the current job has passed, the remaining jobs are again moved to CFS and executed there.

Policy: `OL_POL_DROP` The drop policy removes the required cutback from the last jobs in the schedule. For a utilization of $U_p = 1.2$ this would cut back the last 4 jobs of each task, resulting in a cutback of $4 \cdot (30\,\text{ms} + 90\,\text{ms}) = 480\,\text{ms}$ and the last 8 jobs in the schedule set to zero scheduled execution time. The remaining 20 ms would be cut back from the job previous to the 8 jobs that were cut back to zero.

Depending on the submission order, this might either be a long or a short job, resulting in the short job being cut back to 10 ms or the long job to 70 ms. This does effectively shorten the sum of execution times in the schedule to the available amount of time. When ignoring deadlines, this would yield a feasible schedule, but as jobs are still bound by their deadline, they can only be moved so they finish at their deadline.

This causes the schedule, in case of $U_p = 1.2$, to only be cut back by one period or 100 ms. The remaining 400 ms are still in front of the schedule in an interval that is already in the past.



(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

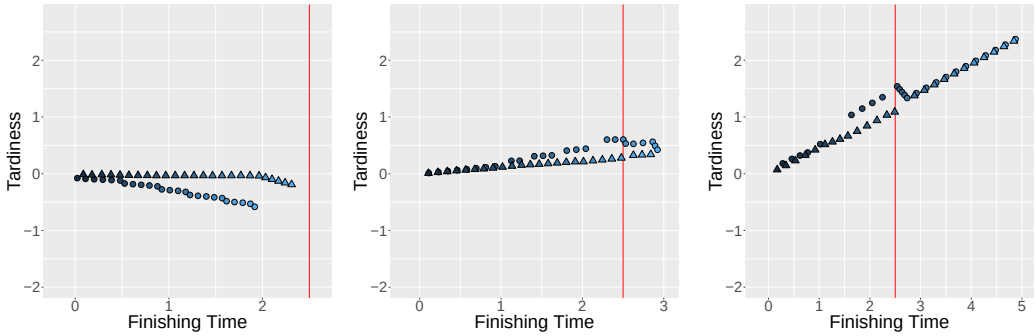
Figure 5.11: Policy: `OL_POL_DROP`

The jobs in the beginning of the schedule shown in Figure 5.11(b) are moved to `EDF Recovery`. The jobs then execute on the scheduled execution time of the remaining jobs in the `ATLAS` band. At some point in time, they reach the jobs that were cut back to zero. At this point, they are moved to `EDF Recovery` and execute in `EDF Recovery` in order of increasing deadlines. When the latest deadline has passed, all jobs are moved to `CFS` and continue their execution there.

Policy: `OL_POL_LAXITY` With the laxity-based policy, the required cutback is distributed proportional to jobs' laxity. Jobs with a higher laxity get cut back more than jobs with a lower laxity. For example, a system with only two jobs $J_1 = (e_1 : 3,\ d_1\ 5)$ and $J_2 = (e_2 : 4,\ d_2 : 5)$ would be overloaded by 2 time units. job J_1 would have a laxity of 2 and job J_2 a laxity of 1. The sum over all laxities would thus be 3. Therefore, job J_1 would be cut back by $2 \cdot {}^2/_3$ and job J_2 by $2 \cdot {}^1/_3$, resulting in scheduled execution times of $\hat{e}_1 = 1{}^2/_3$ and $\hat{e}_2 = 3{}^1/_3$.

A problem with this policy is that shorter jobs are inherently disadvantaged by it. If a short job and a long job have the same deadline, the short job will always be cut back more than the long job, as it has a higher laxity.

Also, the same problem as with the drop policy exists: later jobs are cut back more, as they have a higher laxity. This ignores that jobs just cannot be moved far enough into the future, as they are bound by their deadline.



(a) $U_p = 0.95$ (b) $U_p = 1.2$ (c) $U_p = 2.0$

Figure 5.12: Policy: `OL_POL_LAXITY`

Those shortcomings explain the results in Figures 5.12(b) and 5.12(c). Jobs are cut back more the later they appear in the schedule. While the first jobs can still finish in the time assigned to them in the `ATLAS` band, later jobs have to use their successors time to finish. The short jobs at the end of the schedule are cut back to zero scheduled execution time and can thus only execute in `EDF Recovery`.

That shorter jobs are inherently disadvantaged, can be seen as the jobs of task 1 having a higher tardiness compared to jobs of task 2. After the latest deadline, all jobs are scheduled in `CFS`.

6 Conclusion

In this work, I presented different methods for handling overload with the ATLAS real-time scheduler. As one part of this work, policies for handling overload when it has already occurred were examined. As the other part, facilities for preventing a system from becoming overloaded in the first place were implemented.

In the kernel infrastructure, different policies were implemented that cut back an overloaded schedule to transition it back into a state of being non-overloaded. Furthermore, a system call was realized, that exports information about the current load situation from the kernel to userspace.

The in-kernel infrastructure uses an overloaded schedule, calculates by how much it is overloaded and distributes this overload as a cutback among all jobs on the run queue. How this distribution is done can by controlled by selecting one of five different policies.

These policies represent different heuristics for valuing jobs in a system. While the fixed policy is mostly of theoretical nature, the proportional policy might prove beneficial with jobs that work less than submitted. The drop policy does not work well with a periodic task set, but might be suited when many jobs have a shared deadline. The laxity based policy did also not work too well with periodic tasks but could work better in a system that is only transiently overloaded. Lastly, the fair policy showed to be well applicable to prioritize short jobs of a periodic task over longer ones.

In the ATLAS runtime, I implemented a load manager, that is able to use the information provided by the aforementioned system call to enqueue work into ATLAS without overloading the schedule. It provides a convenient interface, that might in the future be combined with ATLAS' work queues. The load manager can make use of two different modes of operation: functional alternatives and iterative refinement.

Functional alternatives might for example be used when applications can start alternative workloads with different execution times. The load manager can choose an alternative, that fits the current schedule, execute it, and return the delivered result to the calling task.

Iterative refinement functions can be used, when an application knows, that it needs a result at a certain point in time but can handle different levels of precision for this result. The load manager then runs the iterative refinement function for as long as possible until the deadline. In this period, it adapts the submitted execution time of the iterative refinement function via the system call `atlas_update`.

The load manager was evaluated and showed, that it does not cause other jobs, that are already in the system to miss their deadlines. This holds for the functional alternatives as well as the iterative refinement mode. Work started in either of the two modes of

operation are themselves able to meet their given deadlines. The load manager is thus suitable to prevent a system from becoming overloaded in the first place.

Glossary

aperiodic An *aperiodic* task emits only one job. 15, 16, 21, 28, 33

clairvoyant A scheduling algorithm is called *clairvoyant* if, at time 0, it knows about all tasks that will be scheduled 11

feasible A schedule is said to be *feasible* if all tasks that are schedueled complete within their respective deadline. 32, 33

on-line A scheduling algorithm is called *on-line*, if it is given no information about tasks before they are submitted/released. 11, 36

optimality "[...] an algorithm is said to be *optimal* if it may fail to meet a deadline only if no other algorithms of the same class can meet it." [But11] 33

overloaded See underloaded. 38

periodic A *periodic* task emits jobs regularly, with a given period. 15, 16

sporadic A *sporadic* task emits jobs irregularly, with a given minimum inter-arrival time. 15, 16

underloaded "A system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise." [KS92] 38, 81

List of Acronyms

API application programming interface 25

APIC advanced PIC 13, 84

ATLAS Auto-Training Look-Ahead Scheduler 11–13, 21–30, 36, 43–49, 55–58, 60, 62, 64, 68, 70, 71, 73, 78, 86

BIOS basic input/output system 62

BWP Blue When Possible 41

CBS Constant Bandwidth Server 19, 21, 38, 39, 41

CFS Completely Fair Scheduler 18–21, 23, 24, 29, 30

CPU central processing unit 11, 13, 14, 17–21, 24–26, 30, 60, 62

DHCP Dynamic Host Configuration Protocol 62

DSS Dynamic Sporadic Server 38

ECET expected case execution time 36

EDF earliest deadline first 16, 22, 23, 32–34, 36, 38–41

EP ExceptionPart 36

EPU Effective Processor Utilization 37

FIFO first-in-first-out 21, 26

FTP File Transfer Protocoll 62

GCD Grand Central Dispatch 21, 25, 26, 28

GED Guaranteed Earliest Deadline 34

GMP GNU Multiple Precision Arithmetic Library 67

GPL GNU General Public License 16

HLF Horizon Load Function 47, 55, 65

IC integrated circuit 13

ILF Interval Load Function 47, 54, 55, 65

IP Internet Protocol 62

ISA instruction set architecture 16

KVM Kernel-based Virtual Machine 60

LAPIC local APIC 13

LRT latest release time 22, 36

LST Latest Start Time 38

MED Robust Earliest Deadline with multiple task rejection 36

MMU memory management unit 13

MP MainPart 36

MUF maximum utilization factor 36

OS operating system 11, 13, 14, 16

PARSEC Princeton Application Repository for Shared-Memory Computers 63

PIC programmable interrupt controller 13, 83

PIT programmable interrupt timer 13

POSIX Portable Operating System Interface 19, 21

PXE Preboot Execution Environment 62

QoS Quality of Service 11, 12, 15

RAM random access memeory 62

RED Robust Earliest Deadline with single task rejection 33, 34, 36

RMS rate monotonic scheduling 16, 34

ROBUST Resistance to Overload By Using Slack Time 36, 37

RR round-robin 21

RTO Red Tasks Only 40, 41

SMP symmetric multiprocessing 19

STL Standard Template Library 43

TAFT Time-Aware Fault-Tolerant 36

TBS Total Bandwidth Server 38

TP TaskPair 36

WCET worst case execution time 11, 16, 21, 34, 36, 38

List of Figures

List of Listings

List of Tables

Bibliography

[AB98] Abeni, L. and Buttazzo, G. Integrating multimedia applications in hard real-time systems. In: *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pp. 4–13. IEEE (1998).

[App09] Apple, Inc. Grand Central Dispatch (2009). `https://web.archive.org/web/20090920043909/http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf`

[App16] Apple, Inc. Dispatch (2016). `https://developer.apple.com/reference/dispatch`

[BC05] Bovet, D. P. and Cesati, M. *Understanding the Linux Kernel*. O'Reilly Media, 3 edition (2005). ISBN 0596005652,9780596005658.

[BG01] Becker, L. B. and Gergeleit, M. Execution environment for dynamically scheduling real-time tasks. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, pp. 13–16 (2001).

[BH93] Baruah, S. and Haritsa, J. R. *ROBUST: a hardware solution to real-time overload*, volume 21. ACM (1993).

[BKM+91] Baruah, S., Koren, G., Mishra, B., Raghunathan, A., Rosier, L., and Shasha, D. On-line scheduling in the presence of overload. In: *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pp. 100–110. IEEE (1991).

[BKM+92] Baruah, S., Koren, G., Mao, D., Mishra, B., Raghunathan, A., Rosier, L., Shasha, D., and Wang, F. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, volume 4(2):pp. 125–144 (1992). `http://dx.doi.org/10.1007/BF00365406`

[BL11] Bienia, C. and Li, K. *Benchmarking modern multiprocessors*. Princeton University USA (2011).

[BS93] Buttazzo, G. C. and Stankovic, J. A. *RED: Robust earliest deadline scheduling*. University of Massachusetts, Department of Computer Science (1993).

[BS95] Buttazzo, G. C. and Stankovic, J. A. Adding robustness in dynamic preemptive scheduling. In: *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, pp. 67–88. Springer (1995).

[But05a] Buttazzo, G. C. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, volume 29(1):pp. 5–26 (2005). ISSN 1573-1383. doi:10.1023/B:

TIME.0000048932.30002.d9.
http://dx.doi.org/10.1023/B:TIME.0000048932.30002.d9

[But05b] Buttazzo, G. C. *Soft real-time systems: predictability vs. efficiency*. Series in computer science (New York, N.Y.). Springer, 1 edition (2005). ISBN 0-387-23701-1,9780387237015,9780387281476,0387281479.

[But11] Buttazzo, G. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media (2011).

[Ces14] Cesati, M. Overview of the Linux Scheduler Framework (2014).
http://retis.sssup.it/rts-like/cesati.pdf

[GN02] Gergeleit, M. and Nett, E. Scheduling transient overload with the taft scheduler. *GI/ITG specialized group of operating systems* (2002).

[Hof15] Hoffmann, H. JouleGuard: energy guarantees for approximate applications. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 198–214. ACM (2015).

[HSC+11] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In: *ACM SIGPLAN Notices*, volume 46, pp. 199–212. ACM (2011).

[IEE08] IEEE Standards Association and others. IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell*, pp. 1–3874 (Dec 2008). doi: 10.1109/IEEESTD.2008.7394902.

[KS92] Koren, G. and Shasha, D. D^{over}: an optimal on-line scheduling algorithm for overloaded real-time systems. In: *Real-Time Systems Symposium, 1992*, pp. 290–299. IEEE (1992).

[KS95] Koren, G. and Shasha, D. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In: *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pp. 110–117. IEEE (1995).

[Lee15] Leemhuis, T. Die Neuerungen von Linux 4.0 (2015).
https://heise.de/-2600242

[LL73] Liu, C. L. and Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, volume 20(1):pp. 46–61 (January 1973). ISSN 0004-5411. doi:10.1145/321738.321743.
http://doi.acm.org/10.1145/321738.321743

[LNL87] Lin, K.-J., Natarajan, S., and Liu, J. W.-S. Imprecise results: Utilizing partial computations in real-time systems. In: *IEEE Real-Time Systems Symposium, San Jose, CA* (1987).

[Loc86] Locke, C. D. Best-effort decision making for realtime scheduling. *Ph. D. thesis, Department of Computer Science, Carnegie Mellon University* (1986).

[Ric99] Richling, M. Approximate Pi via a beautiful, but inefficient, formula. (1999). `https://www.mitchr.me/SS/exampleCode/GMP/piWallis.c.html`

[Roi13] Roitzsch, M. *Practical Real-Time with Look-Ahead Scheduling*. Ph.D. thesis, TU Dresden (2013).

[SA00] Stappert, F. and Altenbernd, P. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, volume 46(4):pp. 339–355 (2000). ISSN 1383-7621. doi: http://dx.doi.org/10.1016/S1383-7621(99)00010-7. `http://www.sciencedirect.com/science/article/pii/S1383762199000107`

[See13] Seeker, V. Process Scheduling in Linux (12 2013). `https://criticalblue.com/news/wp-content/uploads/2013/12/linux_scheduler_notes_final.pdf`

[TB14] Tanenbaum, A. S. and Bos, H. *Modern Operating Systems*. Pearson, 4th edition (2014). ISBN 013359162X, 9780133591620.

[Tor15] Torvalds, L. Linux (2015). `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/?h=v4.0`

[van11] van Riel, R. [patch -v8 4/7] sched: add yield_to(task, preempt) functionality. (2011). `https://lkml.org/lkml/2011/1/31/423`

[W3T16] W3Techs. Usage statistics and market share of Unix for websites (11 2016). `https://w3techs.com/technologies/details/os-unix/all/all`

[WEE+08] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, volume 7(3):pp. 36:1–36:53 (May 2008). ISSN 1539-9087. doi:10.1145/1347375.1347389. `http://doi.acm.org/10.1145/1347375.1347389`

[Wei16] Weisbach, H. *Multi-Processor Look-Ahead Scheduling*. Master's thesis, TU Dresden (2016).