Master's Thesis

Multi-Processor Look-Ahead Scheduling

Hannes Weisbach

February 3, 2016

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dr.-Ing. Michael Roitzsch

Task

Atlas, developed at the Operating Systems Chair, is an infrastructure to assign CPU time. Central goal of Atlas is to simplify development of real-time applications by relieving the programmer of the burden of providing period and execution time.

Currently, the implementation of Atlas in the Linux operating system only supports uni-processor operation. The goal of this thesis is to add multi-processor support to the existing implementation.

The scientific question of this thesis lies in the design and implementation of a user space/kernel space interface for multi-processor Atlas. The programming paradigm of Atlas and the goal of usability should be maintained and, if necessary, extended with suitable primitives to support parallel execution.

The evaluation should include a comparison of Atlas with an existing Linux multi-processor scheduler, as well as determine the cost of a scheduling decision and the corresponding overhead for applications.

Selbstständigkeitserklärung

I hereby declare, that I have authored this thesis independently, making use only of the specified aids.

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

Dresden, den 3. Februar 2016

Hannes Weisbach

To my parents,
Andrea and Udo Weisbach.

Contents

Introduction

After more than 40 years of research in real-time scheduling the results of that research are predominantly applied to traditional control applications. Poster children for soft-real-time applications such as video decoding, video conferencing and multimedia applications in general are still run as best-effort computations in mainstream, general-purpose computing.

One of the reasons applications do not take advantage of real-time scheduling is that mainstream operating systems[1] such as Microsoft® Windows®, Apple® Mac OS®, Linux®, iOS®, and Android™ have no *conveniently* usable support for applications to communicate their computation requirements. Another reason might be, that the rigid periodic job model, which a large part of the real-time scheduling theory is based on, is not a good fit for applications with large variations in computation requirement.

[1] StatCounter Global Stats. *Top 8 Operating Systems from Dec 2014 to Dec 2015*. (Visited on 01/20/2016).

At least Linux and Mac OS allow applications to specify a recurring computation requirement.[2] How should programmers know how much CPU time the critical part of their application needs? To quote from the *Kernel Programming Guide* of Mac OS:

> "It is very important to make a reasonable *guess* about your thread's workload if it needs to run in the real-time band."[3]

[2] Dario Faggioli et al. *An EDF scheduling class for the Linux kernel*. In: *Proceedings of the Real-Time Linux Workshop*. 2009. Apple, Inc. *Kernel Programming Guide*. 2013. (Visited on 02/02/2016), chapter "Mach Scheduling and Thread Interfaces".

[3] Emphasis mine.

Just guessing seems easy, but it is really not. Consider the diversity of hardware an application runs on and their different processing capabilities. Regardless of whether the application is running on a powerful desktop processor or an energy-efficient mobile processor, the time requested from the system must be sufficient to complete the computation. The result is gross over-estimation of computation requirement, which wastes resources and limits the number of concurrently executing real-time applications.

This is the point where the ATLAS[4] infrastructure offers a solution. Applications using ATLAS are not restricted to the periodic job model. ATLAS offers an easy-to-use API inspired by GCD to submit arbitrary collections of jobs. Instead of worst-case execution times, the abstraction of a workload metric is introduced to allow developers effortless specification of computation requirement in the application domain.

[4] Michael Roitzsch, Stefan Wächtler, and Hermann Härtig. *ATLAS: Look-Ahead Scheduling Using Workload Metrics*. In: *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2013, pp. 1–10. Stefan Wächtler. *Look-Ahead Scheduling*. Diploma Thesis. Technische Universität Dresden, 2012.

The existing implementation of ATLAS lacks support for load balancing and migration. The ATLAS runtime is restricted to serial queues. The goal of this thesis is to design, implement, and analyze support for concurrent queues, load balancing and migration for the Linux implementation of ATLAS.

I give an introduction to uni- and multi-processor scheduling theory, the Linux scheduling core, and Apple Inc.'s Grand Central Dispatch in chapter *Background*. In chapter ATLAS *on Uni-Processor Systems* I recapitulate the core ideas of ATLAS. Finally, in chapter ATLAS *on Multi-Processor Systems*, I describe the design and implementation of multi-processor support in ATLAS, including support for concurrent queues, followed by a characterization of its performance in chapter *Evaluation*.

Background

Let me start this thesis with an introduction of the terminology I use within this thesis, accompanied by an overview of uni- and multi-processor real-time scheduling theory. I examine the Linux scheduler as an example of a practical scheduler implementation and round off this chapter by presenting the Grand Central Dispatch (GCD) parallel programming model as an alternative to explicit thread-parallelism.

Schedulers

In contemporary, multi-programmed desktop operating systems, multiple processes can be in the ready state at the same time. If more than a single process is in the ready state, a decision has to be made which processes to run next. The scheduler employs a scheduling algorithms to make that decision.

A scheduling algorithm is selected to fit the requirements of a particular use case or type of operating system. The literature[1] discriminates between three use cases: batch processing systems, interactive systems and real-time systems.

[1] Andrew S. Tanenbaum. *Modern Operating Systems*. 2nd Edition. Upper Saddle River; NJ 07458: Prentice Hall, 2001.

When multiple processes are ready, selecting any process over any other has both advantages and drawbacks.[2] Thus, there is no "universally best" choice and hence no scheduling algorithm is universally applicable.

[2] For example `fork()`. Selecting the parent saves context-switch overhead but the child is not interactive. On the other hand, assuming that the child will perform an `exec()` may save copy-on-write overhead, if the child runs first.

A scheduling algorithm is designed to meet specific goals with its process selection. Some goals, like fairness, are common to all system types. Comparable processes should receive a comparable share of CPU time, regardless of whether it is in a batch system, an interactive desktop computer or an embedded real-time control system. System-specific goals are, for example, throughput and minimal makespan in batch systems, response time in interactive systems and to meet deadlines or minimizing tardiness in real-time systems.

A SCHEDULE CAN BE COMPUTED AHEAD OF TIME and is only re-enacted when the system is running. Such schedulers are called *offline* schedulers. Using an offline scheduler is a robust way to schedule real-time systems. When an offline schedule for a real-time system is created, it also doubles as feasibility test. The fact that a schedule could be found proves that the task set is feasible. Offline schedulers are not suitable for any interactive system, where the set of running programs changes either as a result of user input or some other external event.

FOR INTERACTIVE SYSTEMS *online* schedulers have to be used. Online schedulers construct a schedule during runtime, with the additional hurdle of limited knowledge. In this thesis I only consider online schedulers.

Real-Time

As soon as computers were integrated in control loops, the point in time a computation completed became critical. In contrast to non-real-time systems, correctness in real-time system not only depends on the logical value of the result, but also on the time the result is available. For every real-time computation, there is a point in time when the result has to be available. This point in time is called the *deadline*. If the result of the computation is available before its deadline passes, the computation is said to *meet its deadline*, otherwise it *misses its deadline*.

Depending on the consequences of a deadline miss, real-time systems are categorized into *soft-*, *firm-*, and *hard-real-time* systems.

In hard-real-time systems all deadlines must be met. Missing even a single deadline is considered an operational failure of the system with possibly catastrophic consequences. Often-quoted examples for real-time systems are digital controllers in process plants, flight controllers in airplanes, radar signal processing,[3] or engine control units in cars.[4]

[3] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[4] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011.

Firm-real-time systems do not require each deadline to be met. It suffices that a certain number of consecutive deadlines are met. A result delivered after the firm deadline has passed has no use, the same as in hard-real-time systems.

Soft-real-time systems allow for an arbitrary number of missed deadlines, in the worst-case a soft-real-time system can miss all deadlines. Results are still useful, even if they are delivered after the deadline has passed. However, deadline misses cause a soft-real-time system to degrade in performance. Typical soft-real-time systems include multimedia applications such as video conferencing, video streaming, and video decoding. Virtual reality applications also have real-time constraints, moving to hard-real-time to prevent virtual reality sickness.[5] But also non-virtual reality games can be considered real-time applications.

[5] Atman Binstock. *Powering the Rift*. May 2015. (Visited on 12/30/2015).

To compare real-time scheduling algorithms, I will select one metric among four candidates I found in the literature.[6] The candidate metrics under consideration are *utilization bound*, *approximation ratio*, *resource augmentation*, and empirical measurement.

[6] Robert I. Davis and Alan Burns. *A Survey of Hard Real-Time Scheduling for Multiprocessor Systems*. In: *ACM Computing Surveys* 43.4 (2011), p. 35.

The utilization bound of a real-time scheduling algorithm A u_A is the maximum number such that all task sets with utilization $u \leq u_A$ can be feasibly scheduled by algorithm A.[7] The advantage of a utilization bound over other metrics is that it can be used as a sufficient feasibility condition which can be efficiently evaluated.

[7] Liu and Layland offer an alternate definition in Chung Laung Liu and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. In: *Journal of the ACM* 20.1 (1973), pp. 46–61. A task set is said to *fully utilize* a processor under a priority assignment, if any increase in execution time of any task makes the priority assignment infeasible. Over all task sets that fully utilize a processor, the minimum utilization is the *least upper bound* of the utilization, or utilization bound, of a static priority scheduling algorithm.

The notion of a utilization bounds extends naturally to multiprocessor systems. A system with m processors, each of unit-capacity, can thus sustain a load of maximum m. No real-time multi-processor scheduling algorithm can have a utilization bound better than m.

The approximation ratio compares the performance of an algorithm A with an optimal, potentially hypothetical, scheduling algorithm.

For example, let m_A be the number of processors required to feasibly schedule a task set τ using algorithm A and m_0 the minimal number of processors required to feasibly schedule the same task set τ.[8] Then, as m_0 approaches infinity, the ratio of $\Re_A = m_A/m_0$ is called the approximation ratio of algorithm A. If $\Re_A$ is finite, A is called *approximate*; if $\Re_A = 1$, A is called *optimal*.[9]

[8] Sudarshan K. Dhall and Chung Laung Liu. *On a Real-Time Scheduling Problem.* In: *Operations Research* 26.1 (1978), pp. 127–140.

[9] Davis and Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems".

Instead of comparing the required number of processors, resource augmentation compares the performance of an algorithm A with an optimal scheduling algorithm by the required relative speed of the same number of processors. Resource augmentation assumes a linear relationship between the increase in processing speed and the decrease in execution time. Assume task set τ on an m-processor system with unit-speed processors. Task set τ is feasible on m processors using scheduling algorithm A, if the processors are running with speed $f(\tau)$. The resource augmentation factor for algorithm A is now given by $f_A = max_{\forall m,\tau}(f(\tau))$. Again, if $f_A = 1$, A is optimal.[10]

[10] Davis and Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems"

The performance of scheduling algorithms can be empirically measured using randomly generated task sets. If a feasibility test is known, the number of generated task sets, that are feasible, can be compared to the number of task sets deemed feasible by the scheduling algorithm. Otherwise, the relative performance of two or more algorithms can be evaluated using generated task sets. Additionally, simulation can be a tool in determining the number of migrations and preemptions required by an algorithm.[11]

[11] Davis and Burns, "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems"

I will use the utilization bound as a comparative metric for scheduling algorithms. On multi-processor systems, the utilization bound is often an expression depending on the number of processors m. For complex expressions, the limit of the expression as m tends to infinity allows an easy comparison of algorithms.

Uni-Processor Real-Time Scheduling

To guarantee that all deadlines in a hard-real-time system can be met, it is analyzed and validated with formal methods. To that end, Liu and Layland introduced in their seminal paper[12] the periodic task model.

[12] Liu and Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"

Computation demand is modelled with tasks executing periodically recurring jobs. Jobs are characterized by a release time r, an execution time requirement e, and a deadline d. The time between release times of subsequent jobs of a task is called the task's period p. The ratio of execution time and period e/p is called the utilization u of a task. The utilization of a task system is the sum of the utilization of all tasks in the system. To cope with variable execution time requirement between successive jobs, a task is modelled with the maximum execution time requirement that can occur, the worst-case execution time (WCET).

Often, the deadline is assumed to coincide with the end of the period, forming implicit deadline task systems.

Furthermore, the Liu and Layland task model assumes a processor has unit-capacity and no overload occurs. Jobs are preemptible at arbitrary points in time and such context-switches occur instantaneously, i.e. require no CPU time. If the number of preemptions are bounded, context-switch costs can be incorporated into the worst-case execution time.

Table 1 summarizes the notation used in this thesis.

Access to a shared resource can be arbitrated in a round-robin fashion to provide equal shares or by using a notion of importance, a priority. In real-time scheduling theory priorities are assigned to tasks or jobs. The scheduler then only needs to pick the task with highest priority or highest priority job to run next.

In the following recapitulation of uni-processor real-time scheduling algorithms, I will use the task set listed in Table 2 as running example to illustrate the delineating characteristics between the presented algorithms.

Table 1: Summary of notation for Liu and Layland tasks.

Symbol	Meaning
m	number of processors
n	number of tasks
τ	set of tasks
τ_i	i-th task
$r_i = r(\tau_i)$	release time of task i
$e_i = e(\tau_i)$	execution time of task i
$p_i = p(\tau_i)$	period of task i
$d_i = d(\tau_i)$	deadline of task i
$u_i = u(\tau_i) = \frac{e_i}{p_i}$	utilization of task i
$u = \sum_{\tau_i \in \tau} u_i$	utilization of task set τ
$s_i(t) = t - d_i - e_i$	slack of task i at time t

Table 2: Task set used as example for uni-processor scheduling algorithms.

Task	(e_i, p_i)
τ_1	$(3, 7)$
τ_2	$(5, 10)$

One of the most prominent uni-processor scheduling algorithms is *Rate Monotonic Scheduling* (RMS),[13] introduced with Liu and Layland's periodic task model, mentioned above. Tasks are assigned priorities according to their period, with smaller periods having higher priority. Such an assignment is called a *static priority* assignment, because priorities do not change during runtime.

[13] Liu and Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment".

The RMS algorithm is an optimal scheduling algorithm within the class of static priority algorithms. This means that if any scheduling algorithm using static priority assignment yields a feasible schedule, then a schedule obtained from RM priority assignment will also be feasible. A drawback of static priority assignment is that the resource utilization depends on the periods of tasks. In the worst-case, the utilization bound is ln 2. The RM priority assignment achieves a utilization bound of 1.0 only for task sets with harmonic periods.

Figure 1: RMS schedule of task set from Table 2 with deadline miss of J_2 at time unit 10.

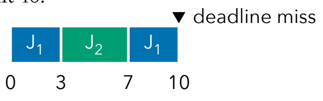
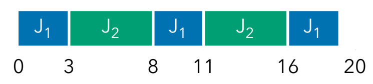


Figure 1 shows the RM-schedule of the exemplary task set from Table 2. With a utilization of ≈0.93 the existence of a feasible RM-schedule is not guaranteed for this task set. At time unit 10 task τ_2 misses indeed a deadline.

To achieve a utilization bound of 1.0 for general task sets, a *dynamic priority* scheduling algorithm, such as *Earliest-Deadline-First* (EDF)[14] is required. Unlike static priority assignment, a task's priority may change during runtime. Priority assignment in EDF is quite intuitive: a job with a close deadline has a higher priority than a job with a deadline further in the future. A task is thus assigned the priority of its current job. EDF is optimal with respect to schedulability in the class of dynamic priority scheduling algorithms.

[14] W.A. Horn. *Some Simple Scheduling Algorithms*. In: *Naval Research Logistics Quarterly* 21.1 (1974), pp. 177–185.

Figure 2: EDF produces a feasible schedule for the task set from Table 2. Between time units 19 and 20 the processor is idle.



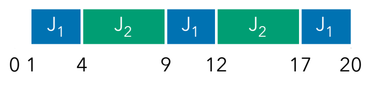
A necessary and sufficient feasibility test for EDF on uni-processor systems is $u \leq 1$. With a utilization of ≈0.93, a feasible EDF schedule is guaranteed to exist for the example task set. The EDF-schedule of the example task set from Table 2 is depicted in Figure 2. Between time unit 19 and 20 the processor is idle for one time unit.

The goal of a real-time scheduler is to meet all deadlines. But there is no gain in completing jobs long before their deadline is up. *Latest-Release-Time-First* (LRT)[15] postpones execution of jobs until

[15] Liu, *Real-Time Systems*.

the latest possible moment such that the deadline can still be met. LRT interchanges release time and deadline of a job and uses EDF to construct a schedule "backwards" from the future to the current time. Hence, LRT is also known as "reverse EDF" or "backwards EDF". By pushing the execution of a job as far into the future as possible, the slack of those jobs can be used to decrease the response time of soft-real-time or best-effort jobs. Same as EDF, LRT is optimal with respect to schedulability, but unlike EDF, LRT requires the knowledge of execution times. LRT is not a priority-driven algorithm since it is non-work-conserving.

Figure 3: LRT schedule is similar to the EDF schedule. The idle-time between time units 0 and 1 shows that LRT is non-work-conserving.



LRT has the same feasibility test as EDF does and so LRT produces a feasible schedule for the example task set from Table 2. The resulting schedule is visualized in Figure 3. In the LRT schedule the processor is idle between time units 0 and 1, although there are active jobs. Hence, LRT is non-work-conserving.

Yet another algorithm for real-time scheduling on uni-processors is *Least-Slack-Time-First* (LST), also known as *Minimum-Laxity-First* (MLF).[16] The *laxity*, or *slack*, of a job is, informally speaking, the amount of time by which the job can "move around". Consider a job scheduled with EDF: the job is executed as early as possible, only preceded by jobs with earlier deadlines. In LRT, the same job is executed as late as possible, only superseded by jobs with later deadlines. The difference between a job's position in an EDF and LRT schedule are part of the jobs laxity. More formally, at any time t, a job with deadline d and and remaining execution time e has a laxity equal to $t - d - e$.[17] Jobs with smaller laxity are given higher priority: the less "wiggle room" a job has to meet its deadline, the more important it becomes.

[16] Yet another name is *Least-Laxity-First* (LLF).

[17] Liu, *Real-Time Systems*

Like LRT, LST requires a-priori knowledge of a job's execution time. The continuous computation of laxity can be computationally expensive. The main difference between LST and EDF/LRT is, that LST has a "more dynamic" priority assignment. In EDF, the priority of a task depends on the task's current job. The task's priority changes only between jobs, but remains constant during execution of any single job. In LRT, a job's laxity (and hence it's priority) remains constant as long as the job is scheduled. When a job is preempted, its laxity decreases linearly. At the same time, the priority of the job increases. This additional dynamism has no advantage for uni-processor scheduling, as EDF is already optimal with respect to schedulability up to utilization 1.0, but it will be important for multi-processor scheduling. This fully-dynamic priority assignment has the drawback that LST can degenerate into time-slicing if two jobs with equal laxity execute concurrently.

Figure 4: LST schedule with degradation to time-slicing between time units 1 to 5 and 14 to 18.

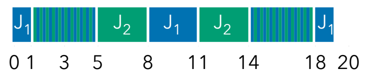


Figure 4 shows the LST schedule for the example task set from Table 2. Between time units 1 and 5 as well as between 14 and 18 LST degenerates into time slicing when both tasks reach a slack of 4. This state only stops when either job has finished execution for the current period. To prevent time-slicing, modified versions of LST have been proposed, which allow laxity inversion.[18] Laxity inversion describes

[18] Sung-Heun Oh and Seung-Min Yang. *A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks*. In: *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*. IEEE. 1998, pp. 31–36.

the situation where the task with least laxity is not scheduled to avoid time-slicing.

Multi-Processor Real-Time Scheduling

Having more processors to execute tasks on does not make real-time scheduling easier. Liu notes:[19]

> Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.

[19] Chung Laung Liu. *Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment*. In: *Space Programs Summary*. Vol. II. The Deep Space Network. 37-60. Jet Propulsion Laboratory, 1969. Chap. 3, pp. 28–31.

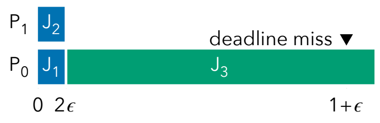
The new dimension mentioned by Liu is the *spatial* dimension. A uni-processor only schedules in the *temporal* dimension, selecting the next process to run. A multi-processor scheduler not only has to answer the question of which process to run next, but “Which process to run next *on which CPU*?” For the rest of this thesis, let m denote the number of processors in a multi-processor system. I will use processor, core, or processor core interchangeably to denote a hardware resource suitable for program execution. I will only consider systems with homogeneous resources. As with uni-processor systems, I assume preemption occurs instantaneously, as does migration.

Table 3: Task set exhibiting Dhall’s effect when scheduled using RMS or EDF on a two-processor system.

Task	(e_i, p_i)
τ_1	$(2\epsilon, 1)$
τ_2	$(2\epsilon, 1)$
τ_3	$(1, 1+\epsilon)$

A straightforward extension from uni-processor priority-driven real-time scheduling to multi-processor scheduling would be to execute the highest priority task on any free processor. If, during run-time, a job arrives with higher priority than any currently scheduled job, the currently running task with the lowest priority is preempted and the newly arrived task is executed. This scheme is called *global scheduling*, because all tasks are kept in a global run queue.

Figure 5: Example of Dhall’s effect when using RMS or EDF.



Using global scheduling, task sets with utilization 1.0 are trivially feasible using a dynamic priority, uni-processor scheduling scheme on a single processor. As Dhall and Liu showed, the minimum achievable utilization using either RMS or EDF on a system with more than one processor may be as low as 1.0.[20]

[20] Dhall and Liu, “On a Real-Time Scheduling Problem”

Consider the task set from Table 3, with the RMS and EDF schedule depicted in Figure 5 as an example for $m = 2$ processors. This example can be extended to a system with m processors with m tasks having the characteristics of τ_1 and τ_2 and one task with the characteristics of τ_3. Letting ϵ tend to zero, the utilization tends towards 1.0.[21] This is also known as Dhall’s effect.

Dhall’s effect causes a task with high utilization to experience interference from a task with low utilization.[22] Realizing the cause of Dhall’s effect, global, static-priority scheduling algorithms with guaranteed utilization bounds were devised, based on the idea of *utilization separation*. Utilization separation algorithms divide a task set into heavy and light tasks, according a utilization value ς, which is a parameter of the algorithm. This threshold is often expressed in the number of available processors, m. All tasks in the heavy set are given the highest

[21] The actual utilization of such a task set for m processors is $u = 2\epsilon m + \frac{1}{1+\epsilon}$. Dhall and Liu, “On a Real-Time Scheduling Problem”

[22] Cynthia A. Phillips et al. *Optimal Time-Critical Scheduling Via Resource Augmentation*. In: *Proceedings of the 29th annual ACM Symposium on Theory of Computing*. ACM. 1997, pp. 140–149. Shelby Funk, Joel Goossens, and Sanjoy Baruah. *On-line Scheduling on Uniform Multiprocessors*. In: *Proceedings of the 22nd Real-Time Systems Symposium*. IEEE. 2001, pp. 183–192.

priority. The tasks in the light set are assigned priority according to a secondary algorithm.

RM-US[$m/3m-2$][23] uses RMS to prioritize light tasks. Its utilization bound is $\frac{m^2}{3m-2}$, and approaches 33 % for large m. The algorithm is parameterized with a utilization threshold of $\varsigma = \frac{m}{3m-2}$ to separate light from heavy tasks. Lundberg proved the optimal utilization threshold for RM-US is in fact $\varsigma = 0.374$ as m tends to infinity.[24] Andersson and Jonsson showed that the upper bound on the utilization bound is $\sqrt{2} - 1 \approx 0.41$ for global static priority-driven scheduling algorithms.[25]

SM-US[$2/3+\sqrt{5}$][26] uses the slack monotonic priority assignment scheme for light tasks. With a utilization bound of 0.382 SM-US improves upon RM-US, not yet achieving the optimal utilization bound of 0.41. Andersson, Baruah, and Jonsson showed that there can be no static-priority multi-processor scheduling algorithm, global or otherwise, with utilization bound better than 0.5.[27]

Unlike on uni-processor systems, using dynamic priority assignment for multi-processor scheduling does not yield a stronger utilization bound over optimal static-priority based algorithms. Srinivasan and Baruah showed that no priority-driven multi-processor scheduling algorithm can achieve a utilization bound better than $\frac{m+1}{2}$.[28]

Consider a task system of $m + 1$ identical tasks, each of which has an execution time requirement of $1 + \epsilon$ and a period of 2, ϵ being an arbitrarily small, positive number. A dynamic priority-driven algorithm must assign each job a priority. Remember that, once a priority has been assigned to a job, the priority remains fixed for the duration of the job.[29] Which ever job is assigned the lowest priority will miss its deadline. Figure 6 depicts a possible schedule for $m = 2$.

A dynamic priority algorithm achieving the upper bound on the utilization bound of Srinivasan and Baruah of $\frac{m+1}{2}$ is FPEDF.[30] FPEDF is an improvement upon the utilization separation-based EDF-US[$m/2m-1$].[31] Baruah was able to prove EDF-US worked correctly with a utilization threshold ς of 0.5.[32]

[23] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. *Static-priority scheduling on multiprocessors*. In: *Proceedings of the 22nd Real-Time Systems Symposium*. IEEE. 2001, pp. 193–202.

[24] Lars Lundberg. *Analyzing Fixed-Priority Global Multiprocessor Scheduling*. In: *Proceedings of the Eighth Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2002, pp. 145–153.

[25] Björn Andersson and Jan Jonsson. *The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%*. In: *Proceedings of the 15th EuroMicro Conference on Real-Time Systems*. July 2003, pp. 33–40.

[26] Björn Andersson. *Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%*. In: *Principles of Distributed Systems*. Springer, 2008, pp. 73–88.

[27] Andersson, Baruah, and Jonsson, "Static-priority scheduling on multiprocessors"

[28] Anand Srinivasan and Sanjoy Baruah. *Deadline-based scheduling of periodic task systems on multiprocessors*. In: *Information Processing Letters* 84.2 (2002), pp. 93–98.

[29] Otherwise the algorithm is referred to as fully dynamic, for example LST in the uni-processor case.

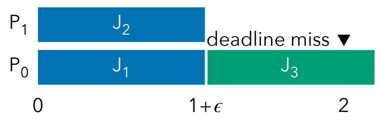


Figure 6: The maximum utilization bound for priority-driven multiprocessor scheduling algorithms is $\frac{m+1}{2}$.

[30] Sanjoy Kumar Baruah. *Optimal Utilization Bounds for the Fixed-priority Scheduling of Periodic Task Systems on Identical Multiprocessors*. In: *IEEE Transactions on Computers* 53.6 (2004), pp. 781–784.

[31] Srinivasan and Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors".

[32] In my opinion, FPEDF should have been called EDF-US[0.5], showing that it is an existing algorithm with improved parametrization.

ANOTHER APPROACH TO SCHEDULE A TASK SET on a multi-processor is to assign tasks to processors and then schedule each processor locally. This approach is called *partitioned scheduling*. Scheduling each processor locally is appealing, because efficient, well-known, and optimal scheduling algorithms can be used. Optimally assigning tasks to processors can be reduced to the bin packing problem, which is NP-complete. An optimal assignment of items, the tasks, to bins, the processors, requires a minimal number of bins. Since the number of processors in a machine is usually fixed, a task system is not feasible if the minimal number of bins of an assignment exceeds the number of available processors.

Task sets are usually small, which makes solving the bin-packing problem optimally easier. Although efficient algorithms, to solve the pin-packing problem optimally, are known,[33] those algorithms are still too slow to apply them on-line. Heuristics are employed to find a near-optimal solution. To perform partitioning, tasks are first sorted,

[33] Richard E. Korf. *A New Algorithm for Optimal Bin Packing*. In: *AAAI-02 Proceedings*. 2002, pp. 731–736

most commonly in order of decreasing utilization, and subsequently assigned to a processor using a bin-packing heuristic such as *First-Fit*, *Next-Fit*, *Best-Fit*, or *Worst-Fit*.[34]

[34] Dhall and Liu, "On a Real-Time Scheduling Problem"; Yingfeng Oh and Sang H. Son. *Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem*. Tech. rep. CS-93-24. 1993. Almut Burchard et al. *New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems*. In: *IEEE Transactions on Computers* 44.12 (1995), pp. 1429–1442. Omar Ulises Pereira Zapata and Pedro Mejıa Alvarez. *EDF and RM Multiprocessor Scheduling Algorithms: Survey and Performance Evaluation*. In: *Seccion de Computacion Av. IPN* 2508 (2005).

A task fits on a processor if the sum of the utilization of the new task and the utilization of all tasks already placed on the processor does not exceed the capacity of the processor.

First-Fit allocation places a task on the first processor where it fits. If no such processor exists, the task is assigned to a new processor.

Next-Fit allocation remembers the processor to which the last task was assigned. Next fit assigns tasks to the same processor as long as they will fit, allocating to an empty processor if a task does not fit. The Next fit strategy does not revisit processors once they were considered at capacity and a new processor is allocated.

The Best-Fit strategy places a task on a processor such that the capacity remaining after the task is assigned to the processor is minimized. If more than one such assignment exists, the processor with the smallest index is chosen. If the task does not fit on any processor, a new processor is allocated.

The Worst-Fit strategy is the opposite of Best fit. Instead of minimizing the remaining capacity after the task is placed on a processor, the remaining capacity is maximized.

For arbitrary task sets, the underlying bin-packing problem limits the maximal utilization bound to 0.5 for partitioned scheduling.[35]

[35] Karthik Lakshmanan, Ragunathan Raj Rajkumar, and John P. Lehoczky. *Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors*. In: *21st Euromicro Conference on Real-Time Systems*. IEEE. 2009, pp. 239–248.

BEING ABLE TO USE ONLY HALF of the processing capability of a machine is unsatisfactory and inefficient. As Funk et al. note, under the assumption that migration and context switching are free,[36] a task set is theoretically feasible if (1) the load of the task set does not exceed the total machine capacity, (2) each task's period or deadline is larger than it's execution time requirement, and (3) preemption and migration is allowed.[37]

[36] i.e. they occur instantaneously

[37] Shelby Funk et al. *DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling*. In: *Real-Time Systems* 47.5 (2011), pp. 389–429.

Using this model, Funk et al. argue that it is possible to reschedule jobs after time ϵ. As $\epsilon \rightarrow 0$, each task appears to be running continuously at a rate necessary to meet its deadline. This model has been termed *fluid* scheduling model.[38] With the fluid scheduling model, a task set is feasible, if the capacity of the machine m is less than the sum of the rates of all jobs. In other words, in order to achieve a utilization bound of m, migration at arbitrary points in time and a fully dynamic priority scheme is necessary.[39]

[38] Anand Srinivasan et al. *The Case for Fair Multiprocessor Scheduling*. In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE. 2003, 10–pp.

[39] John Carpenter et al. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. In: *Handbook of Scheduling: Algorithms, Models and Performance Analysis*. Ed. by Joseph Y-T. Leung. CRC Press LLC, 2000 N.W. Corporate Blvd., Bocy Raton, Florida 33431.: Chapman & Hall/CRC, 2004. Chap. 30, pp. 30.1–30.30.

Fluid scheduling is impractical because of its infinite number of preemptions and migrations. However, real algorithms, tracking the fluid schedule closely, exist and they are capable of constructing feasible schedules for task sets with utilization up to and including m.

To date, the only algorithms solving multi-processor real-time scheduling optimally are based on *proportional fairness*.[40] Proportional fairness forces a task's progress to be very close to its fluid rate. The first algorithm was PFAIR, proposed by Baruah et al.[41] PFAIR chops the time into quanta, requiring each task to be within one quantum of

[40] Funk et al., "DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling"

[41] Sanjoy K. Baruah et al. *Proportionate Progress: A Notion of Fairness in Resource Allocation*. In: *Algorithmica* 15.6 (1996), pp. 600–625.

its fluid rate. However, the drawback of an unnecessarily large number of scheduling decisions and context switches remains.

A number of improvements to PFair have been proposed. ERFair is a work-conserving variant, improving job response times, especially in light-load situations.[42] EKG, proposed by Andersson and Tovar, offers a trade-off between utilization bound and a bounded number of preemptions.[43]

[42] James H. Anderson and Anand Srinivasan. *Early-Release Fair Scheduling*. In: *12th Euromicro Conference on Real-Time Systems*. IEEE. 2000, pp. 35–43.

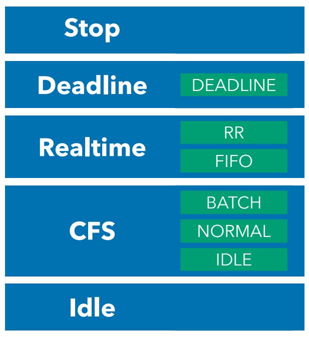
[43] Björn Andersson and Eduardo Tovar. *Multiprocessor Scheduling with Few Preemptions*. In: *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2006, pp. 322–334.

Real-World Case: Linux

After the treatment of real-time scheduling theory in the previous section, I will examine how a real-world, commodity operating system implements (real-time) scheduling. As an example I chose Linux, because Atlas is implemented in Linux. Hence, this section will double-feature to explain the mechanics used in the Linux scheduling core to understand the implementation of Atlas as explained in chapters Atlas *on Uni-Processor Systems* and Atlas *on Multi-Processor Systems*.

The Linux operating system takes a layered approach with its scheduling framework. Each layer has a lower priority than the layer above. Figure 7 depicts the five layers present in a Linux 4.0 vanilla kernel. Whenever a scheduling decision has to be made, the layers are called in order of decreasing priority until a task is returned.

Figure 7: The five scheduling classes in the Linux scheduling framework and the scheduling policies they implement.



The layer with the highest priority is the *Stop* layer. It cannot be selected to run user processes. Only a single process is running with *Stop*-priority on each run queue. In normal operation, this process is blocked and not considered during scheduling. Once unblocked, the stop-process preempts any running process. This property is used for migration, since a process cannot be migrated when it is currently running. The Stop-process then executes the code to migrate a process.

The *Deadline* scheduling class provides an implementation of the EDF scheduling algorithm for Linux.[44] The Deadline scheduling class enforces *temporal isolation*[45] between tasks by using the *Constant Bandwidth Server*-Algorithm.[46] CBS allows to reserve a fraction of the CPU time for a task whose computing requirement does not easily fit with periodic task models but whose mean requirement for CPU time is known. Consequently, soft-real-time tasks can be scheduled alongside hard-real-time tasks. Under default configuration the Deadline scheduling class uses at most 95% of CPU time of the `root_domain`, leaving at least 5% for lower scheduling classes.[47]

The *Realtime* scheduling class implements the POSIX First-In-First-Out (FIFO) and Round-Robin (RR) policies.[48] A process running with the FIFO policy will never be preempted, except when executing a synchronous blocking system call or voluntarily yielding the CPU. A process running with the RR policy will share CPU time equally with other processes running with RR policy. Both policies, FIFO and RR, are global static priority scheduling algorithms,

[44] Faggioli et al., "An EDF scheduling class for the Linux kernel"

[45] Temporal isolation prevents tasks from influencing other tasks ability to meet their deadlines. In other words, no task can cause another task to miss a deadline.

[46] Luca Abeni and Giorgio Buttazzo. *Integrating Multimedia Applications in Hard Real-Time Systems*. In: *Proceedings of the 19th Real-Time Systems Symposium*. IEEE. 1998, pp. 4–13.

[47] Dario Faggioli, Luca Abeni, and Juri Lelli. *Deadline Task Scheduling*. Linux kernel documentation.

[48] *Standard for Information Technology Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*. In: *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)* (Apr. 2013), pp. 1–3906. doi: `10.1109/IEEESTD.2013.6506091`.

but can be restricted to partitioned scheduling by adjusting a process' affinity mask.[49]

[49] By setting the affinity mask to a subset of the available cores, a hybrid between global and partitioned scheduling can be created.

User processes are usually scheduled by the *Normal* scheduling policy. The Normal policy corresponds to the POSIX scheduling policy `SCHED_OTHER`. The algorithm currently used for the Normal policy is the *Completely Fair Scheduler* (CFS). "CFS [...] models an 'ideal, precise multi-tasking CPU' on real hardware."[50] Ideal multi-tasking describes a model in which each process runs at equal speed, i.e. if there are n processes running, each process receives an equal share of $1/n$-th of CPU time. With this construction, CFS is based on the principle of *fair queuing*,[51] originally invented for network scheduling. The share of received CPU time of a process is represented by the *virtual runtime* metric in CFS. Thus, CFS always picks the process with the lowest virtual runtime.

[50] Ingo Molnár. *CFS Scheduler*. Linux kernel documentation.

[51] John Nagle. *On Packet Switches with Infinite Storage*. In: *Transactions on Communications* 35.4 (1987), pp. 435–438.

The concept of virtual runtime is also used by the Borrowed Virtual Time (BVT) scheduling algorithm.[52] BVT also aims at equally distributing CPU time across all applications as well as providing low-latency for interactive and real-time applications.

[52] Kenneth J. Duda and David R. Cheriton. *Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*. In: *SIGOPS Operating Systems Review*. Vol. 33. 5. ACM. 1999, pp. 261–276.

The scheduling policies *Batch* and *Idle*, also implemented by CFS, are Linux specific. The Batch policy trades decreased interactivity for increased throughput. The Idle policy is used to schedule background processes. Processes scheduled by the Idle policy only run when no other process is ready.

THREADS AND PROCESSES are equivalent entities for the Linux scheduling framework. Threads are implemented in Linux as *light-weight processes* (LWP). Both, processes and threads, are represented by an instantiation of `struct task_struct`. The scheduling framework refers to tasks and processes as scheduling entities, embodied by a `struct sched_entity` embedded in each `task_struct`.[53] A scheduling layer may add additional `sched_entity` structures to hold per-task information relevant to the scheduling algorithm implemented in that layer.

[53] `sched_entity` structures are also used to implement group scheduling in Linux. In that case, they are used to form a hierarchy of processes. Jonathan Corbet. *CFS group scheduling*. In: *LWN* (July 2007).

LINUX USES A HYBRID SCHEDULING scheme when running on multiprocessor machines. Linux maintains a run queue for each CPU core, which is a feature of partitioned scheduling algorithms. Tasks on each run queue are scheduled independently from tasks on all other run queues in the system. While this approach eliminates contention on global data structures, it does not balance load between CPU cores. While one core might be idle another core might have two or more ready processes.

To mitigate such imbalances, whenever a CFS run queue has no ready processes it tries to steal work from other run queues[54] by migrating processes to itself until the load is balanced.[55] Additionally, load balancing is initiated in fixed intervals within a scheduling domain. Dynamic task migration is a feature of global scheduling algorithms. Migrations are not fully dynamic as in real-time scheduling algorithms nor are they only performed on tasks joining the system,

[54] Actually, balancing only takes place within a *scheduling domain* and between *CPU groups*.

[55] Migration is further constrained by CPU-affinity of threads.

leading to the conclusion that Linux uses a hybrid approach between global and partitioned scheduling.

Strictly partitioned scheduling is not suitable for Linux because of its primary use as non real-time operating system. As a desktop operating system the applications run on Linux do not fit the stringent scheme of real-time scheduling models. Instead, desktop and multimedia applications are reactive and alter between CPU-intensive phases, I/O, and idle times.

Contrary to CFS, the Realtime and Deadline scheduling classes in Linux also perform load balancing during normal scheduling decisions and operations. During normal operations, there are two mechanisms by which tasks can be migrated. A task can be *pushed* away by its current run queue, or it can be *pulled* by its new run queue. The difference is in which run queue initiates the migration.

Load balancing and migration in the Realtime scheduling class has the goal that no task is blocked while another task with lower priority is scheduled on another CPU. Put differently, the m currently running tasks are in fact the m currently ready tasks with the highest priorities.

Similarly, for the Deadline scheduling class this means that no task should run when a task with an earlier deadline is ready but not scheduled on any other processor. Or, more formally, the m currently running tasks are in fact the m tasks in the systems with the earliest deadlines which are also currently ready.

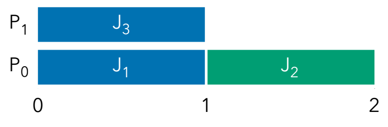
When either scheduling class violates its precondition to run the m tasks which are both ready and have the highest priority according to the respective scheduling class, the system is unbalanced and migration is initiated. By allowing such fine grained migrations the schedule constructed by the partitioned Linux scheduler approaches the schedule constructed by a global scheduler. The implementation of the Linux scheduling framework might cause unnecessary migration as well as fail to perform necessary migrations.[56]

[56] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis. 2011. Chapter 3, page 175

In an unbalanced system, a run queue has *free capacity* when it runs a task with low priority while a task with higher priority, but on another run queue, is not scheduled. Conversely, a run queue is *overloaded* when a high priority task has to wait while on another run queue a task with lower priority is running.

Consider Figure 8. Let's assume that each job is assigned a priority corresponding to its number, i.e. Job 1 has priority 1 and so forth. Let priority 1 be the highest priority. Considering each run queue separately – partitioned – Figure 8 shows a priority-driven schedule. Reconsidering the system under a global schedule, J_3, with low priority, is scheduled before J_2. Under a global perspective the system is not priority-driven. In this case the run queue of processor P_1 has free capacity and the run queue of processor P_0 is overloaded. An overloaded run queue does, in this context, not necessarily imply that the tasks exceed the processing capacity of the CPU. Conversely, free capacity does not necessarily imply that idle time exists in the schedule of a run queue.

Figure 8: An unbalanced system, because low priority job J_3 executes before a job with higher priority, J_2.



A pull balancing operation is initiated by a run queue which has free capacity. Capacity can be generated by a task dropping from its real-time or EDF priority back to the normal priority level. Other reasons are that a task has finished or has been migrated to another CPU, possibly on a user's request.

Load balancing via push operation is initiated by a run queue which is overloaded. Unblocking a previously blocked task is a possible source of overload. The Realtime and Deadline scheduling classes additionally check for overload and initiate pushing on every scheduling decision.

A PER-CPU RUN QUEUE PERFORMING LOAD BALANCING during scheduling decisions has the caveat of having to acquire a second run queue lock. The problem this creates is that resources, the run queue locks, can be acquired in multiple orderings. Maintaining a consistent order of acquiring locks is important, because improper lock ordering may cause deadlocks.

The solution is to potentially drop the lock of the current run queue, so that both locks can be acquired in a consistent order. This solution has the drawback that dropping a lock introduces a race condition in a critical section. After the balancing operation is complete, the preconditions of the critical section have to be re-checked if they still hold. For run queue locks this entails checking that no scheduling layer of higher priority has now a ready task. If so, the scheduling decision has to be aborted and the selection of the next task has to be re-started from the beginning.

Linux uses the addresses of locks to define a locking order. Locks with lower addresses have to be taken first, i.e. when holding lock L_1 at memory location m_1, lock L_2 at memory location m_2 may only be acquired if and only if $m_1 < m_2$.

A SCHEDULING CLASS IN THE LINUX SCHEDULING FRAMEWORK implements an interface consisting of 24 functions,[57] half of which are optional.[58] Merely seven functions are documented.[59] I recapitulate callbacks necessary for general scheduling classes and those important for ATLAS in particular.

[57] As at Linux 4.2

[58] Seven functions deal with SMP support; six of them are optional.

[59] Molnár, *CFS Scheduler*.

`enqueue_task` puts the scheduling entity which is passed as argument in the run queue of the scheduling class. `enqueue_task` is usually called when a blocked thread or process becomes ready, but it is also called when changing a tasks scheduling class or migrating a task.

`dequeue_task` is the inverse operation of `enqueue_task`. It removes a blocked task from the run queue. Additionally, it also used switching a scheduling class and during migration.

`yield_task` is called when a task is voluntarily giving up the CPU. Conceptually, the scheduling entity is put at the end of the run queue.

`check_preempt_curr` determines if a task that just unblocked should preempt the currently running task.

`pick_next_task` returns the next task to run during scheduling decisions. This function is called for each scheduling class until a valid `task_struct` is returned. A return value of `NULL` indicates that a scheduling class has no ready tasks. A return value of `RETRY_TASK`[60] is used to signal the scheduling framework that the selection process should be restarted.

[60] `RETRY_TASK` is a macro expanding to `((void*)-1UL)`. It is used when a scheduling class detects the precondition of calling `pick_next_task` no longer hold. This happens, for example, when the scheduling class dropped the run queue lock to preserve proper lock ordering. After reacquiring the run queue lock, a higher priority scheduling class now has a ready task.

`put_prev_task` stops a running task. This function is used when a new task is selected, during migration, and when the scheduling class of a task is changed.

`select_task_rq` returns the CPU on which a task should run. This is only done to place a task after wake up or when a new task is spawned.

`migrate_task_rq` informs the scheduling class that the `task_struct` passed as argument is about to be migrated to another CPU. The CPU number is also passed as an argument.

`set_cpus_allowed` notifies the scheduling layer that a process' affinity mask has changed.

`set_curr_task` updates the run queue's current task in case of migrations and changes of scheduling class.

`task_tick` is the point where Linux feeds a time base into scheduling classes. `task_tick` is called for the scheduling class of the currently running scheduling entity.

`switched_from` is called when a process changed its scheduling class. It is called for the previous scheduling class.

`switched_to` is called when a process changed its scheduling class. It is called for the new scheduling class. The new scheduling class can determine if it is necessary to preempt the currently running task.

`update_curr` updates the runtime statistics of the currently running task. Updates to statistics are performed only if up-to-date values are required.

Grand Central Dispatch

In the last 10 years parallel computing became available in all major form factors,[61] even tablets and smartphones are featuring quad- and octa-core processors.[62] At the same time, software cannot make effective use of this increase in parallelism. Flautner et al. found that "using more than two processors is not likely to yield great improvements", when analyzing response times of desktop application on multi-processor machines.[63] Flautner et al. note that most application had thread-level parallelism of under 1.4. 10 years later, Blake et al. conducted a similar study finding that 2 to 3 cores "are more than adequate for most applications and that the GPU often remains under-utilized".[64] Blake et al. conclude that threads are rather used to structure programs than gaining parallelism.

[61] Herb Sutter. *Welcome to the Parallel Jungle*. In: *Dr. Dobb's Journal*. (2012). (Visited on 10/29/2015)

[62] Qualcomm Technologies, Inc. *Snapdragon 810*. (Visited on 10/29/2015)

[63] Kristián Flautner et al. *Thread-level Parallelism and Interactive Performance of Desktop Applications*. In: *SIGOPS Operating Systems Review* 34.5 (2000), pp. 129–138.

[64] Geoffrey Blake et al. *Evolution of Thread-Level Parallelism in Desktop Applications*. In: *SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 302–313.

Desktop and office applications do not lend themselves well for parallelization using the fork-join model found, for example, in OpenMP or Cilk. In the task parallel model, applications are decomposed in self-contained work items, which are then distributed and executed by threads or processes.

When implemented by application code, both systems have the drawback of not knowing the effectively available amount of CPU cores. For one, a general desktop application can be run on a wildly diverse set of hardware. But even if the type and number of processors in the system can be ascertained, the application has no knowledge about the computing requirements of other software running concurrently.

Consequently, the execution contexts used to perform the computation represented by work items have to be under the control of a system component which both has knowledge of the hardware configuration as well as of computation requirements of other applications. This system component is the operating system itself.

With Grand Central Dispatch[65] (GCD) Apple Inc. presented an approach to shift the responsibility of managing threads away from applications to the operating system. As a result application developers need to write less code and can more efficiently and effectively utilize a multi-processor systems. GCD is based on the thread pool design pattern. The number of active worker threads in the pool of each application is continuously adapted for optimal performance of the whole system.

[65] Apple, Inc. *Grand Central Dispatch (GCD) Reference*.

Work items, called *blocks* by Apple, are expressed using a language extension to C, C++, and Objective-C which makes code and associated data a first-class citizen of the language, much like lambdas in C++11.

Figure 9 shows how a block is defined and used. A block is initiated by a caret ^, followed by a compound statement containing the code of the block. Variables available in the lexical scope where the block is defined are available inside the block; this is called *capturing* a variable. When a variable is captured, a copy of it is created and associated with the block, so that the original value can be changed without affecting the copy contained in the block.

Figure 9: Block object example.

```
#include <dispatch/dispatch.h>
#include <iostream>

void invoke(void (^block)()) { block(); }

int main() {
  int a = 5;
  auto block = ^{
    std::cout << a << std::endl;
  };
  a += 1;
  std::cout << a << std::endl; // prints '6'
  invoke(block); // prints '5'
}
```

When the value of the variable `a` is changed in line eleven, the copy of `a` captured by creation of the block `block` remains unchanged. A block's type is spelled similar to the type of a function pointer, exchanging the asterisk ∗ for the caret `^`. Similar to functions and function pointers, a block can have arguments. Line four shows how a block can be invoked using the conventional C function call syntax.

Queues are used to structure execution of work items by developers. GCD offers queues in two flavours: serial and concurrent. Work items submitted to global concurrent queues are dequeued by GCD in FIFO order to be processed by a thread pool concurrently. Work items submitted to a concurrent queue can finish execution in any order. Serial private queues can be used to serialize access to shared data, because work items execute in FIFO order, one at time.

A block is submitted for asynchronous execution using the `dispatch_async` function. The synchronous pendant `dispatch_sync` returns only after the submitted work item has completed execution.

Figure 10 shows how a block is submitted to GCD for asynchronous execution. In line six a reference to a concurrent dispatch queue with `DEFAULT` priority is acquired.[66] In lines seven to nine the block is submitted to the queue.

[66] GCD offers four priority levels which are in descending order: `HIGH`, `DEFAULT`, `LOW`, and `BACKGROUND`.

Figure 10: A simple GCD example.

```
 1  #include <dispatch/dispatch.h>
 2  #include <iostream>
 3
 4  int main() {
 5    int a = 5;
 6    auto queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
 7    dispatch_async(queue, ^{
 8      std::cout << a << std::endl;
 9    }); // prints '5' before or after '6' is printed.
10    a += 1;
11    std::cout << a << std::endl; // prints '6'
12    dispatch_barrier_sync(queue, ^{});
13  }
```

In line twelve a barrier is added to the queue, so that the queue is drained before the main thread, and hence the application, exits. Because the block submitted to GCD is executed asynchronously by a GCD worker thread no assumptions can be made regarding the order of execution or completion of the block submitted to GCD and lines ten and eleven of the main thread.

ATLAS *on Uni-Processor Systems*

This chapter provides background information about ATLAS and highlights major design changes which emerged during porting ATLAS to Linux 4.0. I focus the description primarily on changes to the original implementation, but I also elaborate on details which are important when considering multi-core support for ATLAS in the following chapters. Most of ATLAS' design has been presented before[1] and this chapter draws heavily from this previous work.

[1] Roitzsch, Wächtler, and Härtig, "ATLAS: Look-Ahead Scheduling Using Workload Metrics"; Michael Roitzsch. *Practical Real-Time with Look-Ahead Scheduling*. PhD thesis. Technische Universität Dresden, 2013. Wächtler, "Look-Ahead Scheduling".

The ATLAS *Task Model*

In ATLAS a GCD block corresponds to a job of a real time task. A real time task, in turn, is represented by a serial GCD queue. The first ATLAS prototype had no support for concurrent queues. Formally, ATLAS schedules a finite task set τ of n tasks: $\tau = \{\tau_i | i = 0, \dots, n\}$. The task set τ is not fixed, but may change over the runtime of the system, because the user can start and stop applications at her discretion.

Each task τ_i in ATLAS is a set of k jobs: $\tau_i = \{J_{i,j} | j = 0, \dots, k\}$. I will continue to use i to denote a task and j to denote a job of a task. ATLAS does not use a periodic task model, so the scheduler does not have knowledge of future jobs in form of per-task utilization or otherwise. ATLAS does not assume a minimal inter-arrival time either. ATLAS processes jobs in deadline order,[2] therefore inter-job dependencies can be described within ATLAS by proper selection of deadlines.

[2] Previously, ATLAS processed jobs in FIFO order, like GCD. I discuss why this behaviour was unsuited for ATLAS in section *Concurrent Queues* of chapter ATLAS *on Multi-Processor Systems*.

For non-CPU-bound work, such as I/O, ATLAS supports non-real-time jobs. Non-real-time jobs are handled completely in user space by the ATLAS runtime. The idea of non-real-time jobs is to help programmers express jobs, that do neither have a deadline as such nor have useful metrics. Without non-real-time jobs programmers would have to invent deadline and metrics for such jobs, which stands in contrast to ATLAS' principle of helping programmers. Non-real-time jobs are not subject of this thesis.

THE NOTATION FOR THE LIU AND LAYLAND TASK MODEL, introduced in the *Background* chapter, is not suitable for the ATLAS task model. Therefore, I introduce a new notion of execution time and utilization compatible with jobs and tasks as defined by the ATLAS task model. A summary of the notation is given in Table 4.

Table 4: Summary of notation for ATLAS tasks.

Symbol	Meaning
$J_{i,j}$	j-th job of task τ_i
$r_{i,j}$	submission time of job $J_{i,j}$
$e_{i,j}$	execution time of job $J_{i,j}$
$d_{i,j} = d(J_{i,j})$	deadline of job $J_{i,j}$
$s_{i,j}(t)$	slack of job $J_{i,j}$ at time t
$u_i(t)$	utilization of task τ_i at time t
$e_i(t)$	execution time of task τ_i at time t

Since ATLAS jobs are non-periodic, a job is only characterized by its execution time and deadline. The ATLAS task model does not presume jobs of a task τ_i have a common execution time. Let $J_{i,j}$ denote the j-th job of task τ_i. $e_{i,j}$ denotes the execution time and $d_{i,j}$ the deadline of job $J_{i,j}$. A job is released when it is submitted to the ATLAS scheduler at time $r_{i,j}$.

The limited knowledge of ATLAS makes it hard to calculate execution times or utilizations of ATLAS tasks. The only feasible solution is an approximation based on current knowledge. I define the execution time of an Atlas task τ_i at time t, $e(\tau_i, t)$ as:

$$e(\tau_i, t) = e_i(t) = \sum_{J_{i,j} \in \tau_i} e_{i,j}$$

In absence of a task period, I define the utilization of an Atlas task at time t as ratio of the reserved execution time and the available CPU time:

$$u(\tau_i, t) = u_i(t) = \frac{e_i(t)}{\max_{J_{i,j} \in \tau_i} d_{i,j} - t}$$

The utilization remains constant during execution of a task, but changes if the task blocks, a new job arrives, or a job is cancelled.

The task model of Atlas largely prevents a formal feasibility analysis in exchange for more flexibility when adapting applications to Atlas. No optimal online scheduler exists for a collection of arbitrary jobs with more than one distinct deadline scheduled on more than one processor,[3] which is exactly the flexible task model Atlas provides. This leads to a concept of real-time as a service[4] (RTaaS). In RTaaS the contract between application and scheduler is as follows: as long as the collection of jobs, generated by the application, is feasible, the scheduler guarantees to meet all deadlines. Given a set of jobs, feasible on a platform π with an optimal offline scheduler, techniques such as *resource augmentation*[5] can be used to derive how much faster a platform π' needs to be such that the task set is feasible using a non-optimal online scheduler. In this thesis I restrict myself to the Liu and Layland task model.

[3] Kwang S. Hong and Joseph Y-T. Leung. *On-Line Scheduling of Real-Time Tasks*. In: *Proceedings of the Real-Time Systems Symposium*. IEEE. 1988, pp. 244–250.

[4] Funk, Goossens, and Baruah, "On-line Scheduling on Uniform Multiprocessors".

[5] Phillips et al., "Optimal Time-Critical Scheduling Via Resource Augmentation".

Auto-Training Look-Ahead Scheduling

The advantage of Atlas is that it removes the burden of providing worst-case execution times from developers by predicting execution time requirements from workload metrics. At the same time, Atlas offers a flexible and easy-to-use programming interface inspired by that of GCD.

I will start with a short overview of Atlas' inner workings. The rest of this section contains an in-depth description of relevant parts of the Atlas runtime and scheduler. The accompanying Figure 11 visualizes graphically how Atlas processes jobs.

The Atlas runtime exposes a subset of the GCD interface so application developers are presented with a convenient API to submit jobs $J_{i,j} = (\vec{m}_{i,j}, d_{i,j})$. The GCD-inspired interface allows developers to pass an absolute deadline $d_{i,j}$ and a vector $\vec{m}_{i,j}$ of workload metrics.

The Atlas runtime forwards the workload metrics and its internal state to the per-application prediction component. The predictor uses the workload metrics to predict an approximate execution time $e_{i,j}$ for the job. The job description $J_{i,j}$ is stored by the runtime component until the job has completed execution and its actual execution time is known. After a job has completed, the actual execution time is

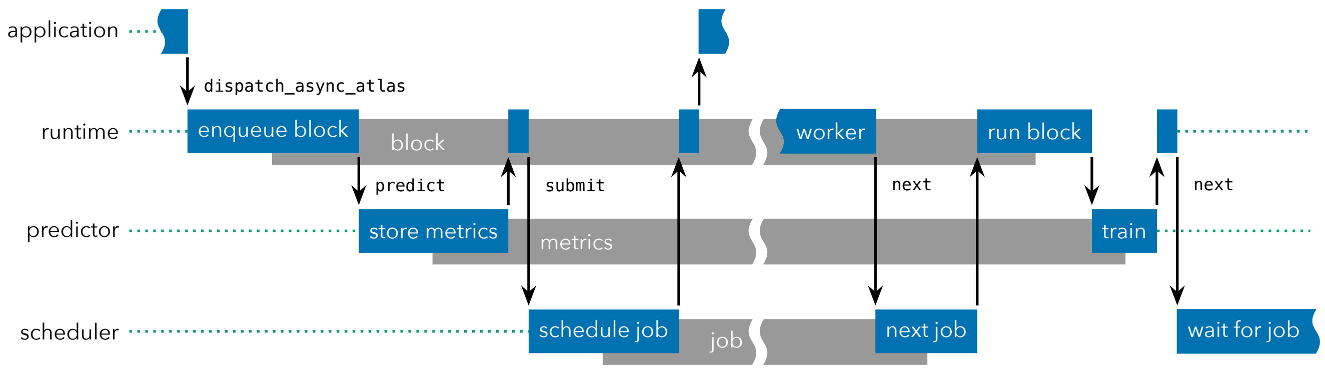


Figure 11: Life cycle of an Atlas job. Adapted from Michael Roitzsch. *Practical Real-Time with Look-Ahead Scheduling*. PhD thesis. Technische Universität Dresden, 2013.

measured and fed back into the prediction mechanism to improve future predictions.

The Atlas scheduling layer receives the modified job description $J'_{i,j} = (e_{i,j}, d_{i,j})$ from the runtime. The Atlas scheduler reserves execution time for each submitted job. Because the scheduler has global knowledge of all applications, Atlas can optimize across processes and predict overload situations.

What's All This Workload Metrics Stuff, Anyhow?[6]

[6] Famous analog integrated circuit engineer Robert Allen Pease commonly used a headline of the form *What's All This <Topic> Stuff, Anyhow?* for his column "Pease Porridge" in the Electronic Design magazine.

The problem of worst-case execution times is that they depend on the hard- and software environment of the machine the real-time application is running on. This dependency might be manageable in a closed system, where the configuration changes only in a controlled way, if at all. It is not suitable for running soft-real-time applications on ever-changing commodity hardware of end-user devices.

Even worse, application developers rarely know how to measure or estimate worst-case execution time, or how to use the tools to do so. Thus extra time and effort has to be spent training them. What those developers *do* have is knowledge from the application domain. This is why workload metrics, which are taken from the application domain, are a more promising solution to specify execution times.

A workload metric is required to positively correlate with the amount of work[7] which has to be performed to complete the job. For example, a workload metric might be the number of iterations required of a loop contained in a job.

[7] i.e. the CPU time required to process the job.

Multiple workload metrics can be used to improve the accuracy of the predicted execution time. Metrics not correlating well with execution times are filtered out by the prediction algorithm automatically. Thus "bad" workload metrics cannot do any harm.[8]

[8] See Roitzsch, "Practical Real-Time with Look-Ahead Scheduling" for a in-depth treatment of the prediction algorithm.

The Atlas *Runtime*

The interface of the Atlas runtime library consists of modified versions of the GCD functions `dispatch_async` and `dispatch_sync`.[9] I extended the Atlas runtime with an even more convenient C++ front end, allowing developers to pass lambda expressions in addition to blocks.[10]

[9] Additionally, the functions `dispatch_async_f` and `dispatch_sync_f` are available. The `_f` versions take a function pointer instead of a GCD block as argument.

[10] Technically, the C++ front end also accepts references to function pointers (`R (*&)(Args)`), references to functions (`R (&)(Args)`) and references to blocks (`R (^&)(Args)`).

As in previous versions of ATLAS, queues and predictors remain orthogonal. This means that the knowledge of the predictor is available to every queue, so there is no necessity to submit a block to the same queue for all its invocations. The ATLAS runtime identifies a block by the pointer to the anonymous function associated with each block. If a function pointer is submitted, it is used directly as an identifier. To identify a lambda the hash of its type index is used.[11] Polymorphic wrappers, such as `std::function` or `std::packaged_task` cannot be used to specify jobs to the ATLAS runtime, because the wrapped target is type-erased. To use polymorphic wrappers with ATLAS, they need to be wrapped in a lambda,[12] like the example in Figure 12. It is up to the developer to make sure that function wrappers submitted that way share a common identity. `std::bind` expressions work with ATLAS as long as the target is a function pointer, block or lambda.

[11] `http://en.cppreference.com/w/cpp/types/type_index`

[12] Or any other type with an identity that the ATLAS runtime recognizes.

```
void dispatch(dispatch_queue queue,
              const std::chrono::steady_clock::timepoint deadline,
              const double *metrics, const size_t metrics_count,
              std::function<void(void)> polymorphic_wrapper) {
  dispatch_async(queue, deadline, metrics, metrics_count,
                 [f = std::move(polymorphic_wrapper)] { f(); });
}
```

Figure 12: Giving polymorphic wrappers their identity back.

Each serial queue in ATLAS is backed by a single worker thread. The scheduler assigns CPU time to the worker thread according to jobs submitted to the queue. The worker thread runs a loop, processing jobs and training the predictor.

The ATLAS *Scheduler*

The ATLAS scheduling class can be thought of as an adaption layer between execution time reservation in the form of jobs and Linux processes and threads as execution context. I will refer to all scheduling entities, processes and threads, henceforth as threads. While technically incorrect, such a simplification improves readability and avoids introducing a new term.[13]

THE INTERFACE TO THE SCHEDULER COMPONENT consists of four system calls.[14]

`submit` notifies the scheduler of a new job. The arguments of the system call are the absolute deadline of the job, the predicted execution time, the thread ID, as returned by `gettid`,[15] of the thread supposed execute the job, and an arbitrary 64 bit number identifying the job.[16] When the ATLAS runtime is used, the thread ID passed will be the thread ID of the worker thread of the queue to which the job was submitted. However, the scheduler interface can be used directly, so any thread ID can be used to implement producer-consumer patterns manually.

[13] The much-loved 'task' is of no help here, since it would only lead to confusion with its overlapping use in real-time scheduling theory.

[14] In the implementation the system calls are named `atlas_submit`, `atlas_next`, `atlas_remove`, and `atlas_update`. For readability and typographic reasons I will omit the `atlas_`-prefix.

[15] Contrary to earlier versions of ATLAS, the target thread has to reside in the same process as the submitting thread. A thread can also submit work to itself.

[16] The job ID is also a new feature. Previous versions of ATLAS required jobs to be processed in FIFO order, requiring jobs to be submitted with monotonically increasing deadlines. Now, the kernel can reorder jobs according to deadline and notify the user space of the job to execute by returning its ID in the `next` system call. If this feature is used, the FIFO-processing property of GCD is violated and jobs must not depend on one another. If in doubt, use monotonically increasing deadlines to force jobs to be processed in FIFO order.

`next` is called to notify the scheduler that a job has been completed. This system call is invoked by worker threads of serial queues. The worker knows what code to execute from the job description stored by the Atlas runtime. The `next` system call returns the 64 bit job identifier, established when submitting a job, of the job supposed to be executed next via out-parameter. This way, the kernel notifies the user space which job should be processed next. While not strictly necessary for serial queues, it will become important for concurrent queues, discussed in chapter Atlas *on Multi-Processor Systems*. If the worker thread has processed the last job, the `next` system call will block until new jobs have arrived.

`remove` causes the scheduler to discard a job from the schedule. The job must not have been started. While the Atlas runtime offers currently no mechanism to cancel jobs, the Atlas scheduler offers that functionality. The idea is that applications can cancel optional work in overload situations. The implementation of this feature in applications and the Atlas runtime is out of the scope of this thesis.

`update` changes the deadline, the execution time, or both of a job. This system call is intended to adapt applications computation requirements in overload situation by reducing the execution time, extending the deadline, or both. Like `remove`, there is currently no Atlas runtime interface to change a job's metrics. The implementation of such a feature is out of the scope of this thesis as well.

The Atlas scheduling layer maintains the jobs submitted to each thread sorted by monotonically increasing deadline. Additionally, Atlas maintains such a list on each run queue[17] with the jobs of all threads on that CPU. This per-run queue job list is the backbone of the Atlas scheduler, similar to the red-black tree of threads in CFS.

[17] For a uni-processor system 'per-run queue' and 'global' data structures are equivalent.

If a scheduling decision needs to be made, Atlas probes the list of jobs from front to back for the job with the earliest deadline, whose thread is ready. This is the thread Atlas selects to run.[18]

[18] Previously, Atlas kept a list of threads, similar to CFS, sorted by the earliest deadline of each thread. This construction turned out to make the implementation of Atlas more complicated than necessary and also had performance drawbacks.

While a job is being processed four things can happen. The thread can block, a new job with earlier deadline arrives, the job finishes meeting its deadline, and the job misses its deadline.

Real-time scheduling theory assumes that tasks do not block. Since Atlas aims for practicality it has to deal with application code that blocks during a job, although it should not. An application is most likely to block on I/O operations, but it is also possible for an application to sleep during a job. If a thread blocks, there is not much that can be done. The Linux scheduling framework will remove the thread from the run queue and a new thread will have to be selected. Atlas will select, again, the job with the earliest deadline, whose thread is not blocked. Once the blocked thread is ready again, `check_preempt_curr` will be called to determine if the currently running process should be preempted.[19]

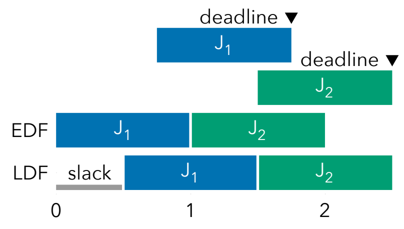
[19] That is, if the currently running process is also an Atlas thread. Otherwise preemption depends on the scheduling class of the currently running process. Processes of lower layers will be preempted, those of higher layers will not.

A new job will only cause a currently running Atlas thread to be preempted if it has an earlier deadline than the earliest deadline of the jobs of the currently running thread. A new job with deadline later than that does not have any influence on the currently running thread. Additionally, the thread, the job was submitted to, must not be blocked.

If the job finishes its execution before the deadline has passed, the job is removed from both, the per-thread list of jobs as well as the per-run queue list of jobs. I will discuss how Atlas deals with missed deadlines in the section *Broken Promises*.

By ordering jobs by increasing deadline, Atlas effectively generates an EDF schedule. However, similar to LRT, Atlas does not process jobs eagerly. Instead, Atlas delays execution until the latest possible moment. Atlas uses the predicted execution time, which is subtracted from the deadline, to find the latest moment processing of a job has to begin. Starting from the job with the latest deadline, working "backwards" in time, Atlas constructs the schedule in reverse order. Reverse construction allows Atlas to let jobs finish either at their deadline or at the beginning of the next job.

Figure 13: How Atlas constructs a schedule.



Consider Figure 13 with jobs J_1 and J_2, each with an execution time of 1 time unit. An EDF schedule of those two jobs would have J_1 finish at time unit 1 and J_2 at time unit 2, both well before their respective deadlines. The schedule Atlas constructs, inspired by *Latest Deadline First*[20] (LDF), starts out with the latest deadline at time unit 2.5. For job J_2 to meet its deadline it has to start at time unit 1.5. Job J_1 cannot execute between time unit 1.5 and 1.75, because J_2 must execute during that interval to meet its deadline. Job J_1 is scheduled such that it completes at the minimum of either its deadline or the start of the next job. Formally, a job's scheduled deadline $d_i^s = \min\left(d_i, d_{i+1}^s - e_{i+1}\right)$, yielding an iterative algorithm to construct the Atlas schedule from right-to-left. In contrast to LDF, Atlas does not have to consider precedence constraints.

[20] Eugene Leighton Lawler. *Optimal Sequencing of a Single Machine Subject to Precedence Constraints*. In: *Management Science* 19.5 (1973), pp. 544–546.

The reclaimed slack can be used to execute a-periodic or non-real-time work in a real-time system,[21] among others. Atlas uses the slack to gain interactivity by handing off CPU time to the CFS scheduling layer. This allows non-real-time applications to get timely service and interactivity from the carefully tuned CFS scheduler, while Atlas is still able to reserve execution time for real-time tasks.

[21] Too Seng Tia. *Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems*. PhD thesis. University of Illinois at Urbana-Champaign, 1995.

The drawback of such a rigid system is obvious. Let us consider again Figure 13. In the current system CPU time from time unit 0 to 0.5 is donated to CFS to run non-real-time tasks. But what happens if no non-real-time task requires execution during the slack-time interval, but instead in the interval from time unit 1 to 1.5? In the current configuration, that CPU time would be lost, because Atlas is non-work conserving.

To handle this situation more gracefully, Atlas has a feature called *pre-roll*. The idea of pre-roll is to allow the front-most Atlas job to start execution, but to compete with all other threads scheduled by CFS. The way Atlas achieves this behaviour is by switching J_1's thread to the CFS scheduling class for the duration of J_1's slack. This way, J_1's thread receives anywhere from all CPU time to no CPU time at all – just as CFS sees fit. In essence, during a job's slack time, its thread is competing with all non-real-time threads in CFS. Pre-roll is activated whenever ATLAS donates slack time to CFS. Since pre-rolled threads have no guarantee to receive any CPU time, execution time received during pre-roll is not accounted against a job's execution time reservation in the ATLAS scheduler.

When the calculated slack is up, at time unit 0.5 in the example of Figure 13, job J_1's thread is switched back to the ATLAS scheduling class and receives the reserved execution time. Depending on how much CPU time a thread received during its pre-roll, the job will finish earlier than its reservation. Although the CPU time received during pre-roll is not accounted to a job's reserved execution time, the CPU time is accounted as thread-runtime, which is used to train the ATLAS predictor.

To come back to our example, let us assume, that no CFS thread is ready during slack interval of J_1 and hence J_1 receives the full 0.5 time units of execution time. With only 0.5 time units of execution time remaining, J_1 will finish at time unit 1. Figure 14 visualizes the schedule for this case. At this point J_1 passes its slack to J_2. Whenever a job can execute during slack time, this time is passed down as slack to later jobs.

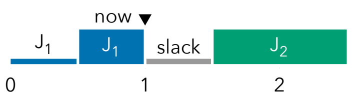


Figure 14: Pre-rolling of J_1 in CFS and passing on of slack to later jobs.

During the interval between time units 1 to 1.5 interactivity is now preserved, because J_2's thread competes with non-real-time threads in CFS. From time unit 1.5 onward the thread of job J_2 is scheduled by the ATLAS scheduling layer until its completion. The earliest time point J_2 can finish is time unit 2, under the assumption that J_2 receives all CPU time during its slack. The latest time J_2 can finish is at time unit 2.5, under the assumption that J_2's thread receives no CPU time during J_2's slack. In either case J_2 meets its deadline at time unit 2.5.

Passing down slack during pre-rolling does not only help interactivity. Pre-rolling also allows ATLAS to compensate errors in predicted execution time and system overheads. System overhead includes the cost of context switches as well as the cost of `next` and `submit` system calls necessary to operate ATLAS.

Broken Promises – Deadline Misses in ATLAS

Jobs can miss their deadlines for a variety of reasons, including an error in execution time prediction, blocking for I/O, self-suspension, or the system being oversubscribed. No matter the reason, ATLAS has to deal with jobs overrunning their deadline. This section describes the different methods ATLAS employs to cope with deadline misses.

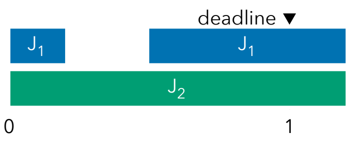


Figure 15: Deadline misses due to blocking and execution time overrun.

ATLAS delineates between two flavours of deadline misses, depicted in Figure 15. The first flavour is caused by a thread blocking during job execution. A deadline miss due to blocking causes a thread to not being able to use the execution time reserved for it. ATLAS does not distinguish between blocking during I/O and self-suspension. Such a scenario is depicted in the upper schedule of Figure 15.

The second kind of deadline miss is execution time overrun, caused by context switching- and scheduling overheads, errors in the execution time prediction,[22] or an application lying about its workload metrics. While the thread uses all the CPU time reserved for it, the execution time is simply not sufficient to complete the job. The second schedule in Figure 15 shows the deadline miss of job J_2 as a result of execution time overrun.

[22] To compensate for errors in the execution time prediction, the Atlas predictor uses overestimation, see Roitzsch, "Practical Real-Time with Look-Ahead Scheduling" for details.

Atlas' handling of deadline misses depends on which kind of deadline miss had occurred.

A deadline miss caused by blocking is not considered to be the application's fault. Therefore Atlas tries to help the application to catch up. The idea is to use the slack of Atlas jobs, so that jobs with deadline misses can *recover* time spent blocked. To that end, Atlas keeps all jobs which missed their deadline by blocking in a separate scheduling band, Atlas *Recovery*. In the slack time donated by real-time jobs, Atlas schedules these jobs in EDF order. I will call this part of Atlas 'EDF Recovery' to avoid confusion with the EDF-implementation in the Deadline scheduling class. The slack time used by this recovery mechanism is lost for pre-rolling Atlas jobs in CFS. As long as the recovery queue is not empty, no task will pre-roll.

A job which missed its deadline by blocking will execute in EDF Recovery only for the amount of time it spent blocking during its original execution time reservation. If the job has not finished when this time is up, the job is removed from EDF Recovery and demoted to CFS, as all jobs with execution time overruns are.

The same mechanisms used to handle deadline misses by blocking are used to handle overload situations in Atlas. In an overload situation, the left-to-right LDF schedule construction used by Atlas will push jobs 'into the past'. Consider the task set in Table 5 and the corresponding schedule in Figure 16 as an example. The schedule at time unit 0 is the initial situation with jobs J_1 and J_2 scheduled. With the release of job J_3, the combined execution time requirement of all three jobs exceeds the available capacity up to time unit 2. Contrary to EDF, deadline misses do not occur in the future. The right-to-left construction of the LDF-schedule pushes the time reservation of job J_1 into the past. A job with a deadline in the past is considered to have missed its deadline. If the job has not received its full execution time, it will be transferred to the EDF Recovery scheduling band. Albeit overload management is out of the scope of this thesis, I will discuss in chapter *Conclusion & Future Work* better ways to handle overload situations.

Table 5: Example task set causing overload when J_3 is released.

Job	r	e	d
J_1	0	0.5	1
J_2	0	1	1.5
J_3	ϵ	1	2

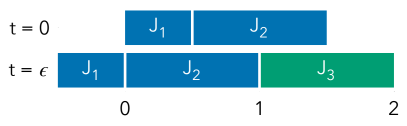


Figure 16: The release of J_3 causes overload. J_1 is 'pushed' into the past and will eventually be demoted to EDF Recovery.

Execution time overruns are considered to be the application's fault. For this reason, a thread which missed a job's deadline by execution time overrun is demoted from the Atlas scheduling class to the CFS scheduling class. Atlas make no effort to support such threads;

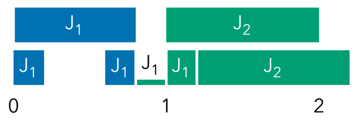
the jobs must complete on a best-effort basis. This part of ATLAS is named 'CFS Recovery' to distinguish it from EDF Recovery, the CFS scheduling class, and pre-rolling in CFS.

OF COURSE, EVERY RULE HAS AN EXCEPTION, and so does the handling of deadline misses in ATLAS. Whenever a job is so much delayed that it runs into the reservation of the next job for the same thread, the thread will receive the reserved execution time with ATLAS priority.[23] The reason for this change is threefold. (1) By submitting a job, a proper request for execution time was made. (2) Since the CPU time for later jobs was properly requested, scheduling a delayed thread does not interfere with other threads or tasks. (3) Scheduling a delayed task might help in catching up.

[23] In Wächtler, "Look-Ahead Scheduling" a job only got promoted to the ATLAS scheduling class again, when it finally caught up.

Figure 17 exemplifies this situation. The upper row shows the planned schedule for task τ_1, with jobs J_1 and J_2. The lower schedule shows how the reservations are enforced in the face of deadline misses. Let us assume that J_1 is able to execute for 0.2 time units, before it is forced to block for 0.4 time units. After J_1 is ready again, its thread runs until J_1's deadline at time unit 0.8. Since J_1 was blocked, its thread is enqueued in EDF Recovery to allow it to catch up the 0.4 time units lost during blocking.

Figure 17: ATLAS threads receive reserved execution time, even if the previous job is not finished yet.



Normally, J_2 could use the slack in the interval between time units 0.8 and 1 to pre-roll. In this case, the slack time is donated to the non-empty EDF Recovery queue to allow the late jobs to catch up.

At time unit 1, J_1 runs into the reservation of J_2. At this moment the thread executing J_1 is promoted again into the ATLAS scheduler class. J_1 executes during the time reserved for J_2, as indicated by the green color.

By time unit 1.2 J_1 finishes and J_2 begins execution.

Because J_1 executed during J_2's execution time reservation, J_2 might in turn miss its deadline. Because J_1's execution time during the reservation for J_2 is not accounted to J_2, at the deadline of J_2 at time unit 2, it looks like J_2 was blocked for 0.2 time units. Thus, J_2 is also eligible for execution in EDF Recovery.

The same donation of execution time from later jobs to earlier jobs that missed their deadlines is employed when the deadline miss was caused by execution time overrun.

Figure 18: How ATLAS integrates into the Linux scheduling framework.

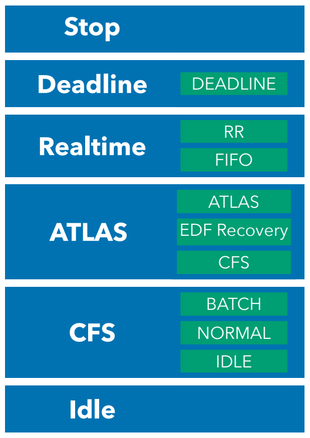


Figure 18 illustrates the integration of ATLAS into the Linux scheduling framework. The ATLAS policy[24] is used for real-time ATLAS jobs. The EDF Recovery policy is used by ATLAS to catch up jobs that have missed their deadline due to blocking, by donating slack time to them. The CFS policy in the ATLAS scheduling class is used to (1) catch up jobs in CFS Recovery and (2) pre-roll ATLAS jobs in the CFS scheduling class if the EDF Recovery scheduling band is empty.

[24] No ATLAS policy can be selected from user space by means of `sched_setscheduler`. A job has to be submitted using the `submit` system call.

Related Work

The WCET is often an outlier, very far from the mean execution time. Scheduling jobs using the WCET results in under utilization of the system, or unnecessary over-provisioning of hardware resources. The multiframe task model[25] generalizes the periodic task model by using a vector of execution times. Consecutive jobs of a task use consecutive elements from the vector as their execution time, wrapping to the start of the vector if the last element has been reached. Nevertheless, period and deadline are still constant for all jobs.

Even greater flexibility offers the generalized multiframe task model,[26] where periods and deadlines are represented as vectors as well.

Atlas increases the flexibility of the task model further by removing periods or minimal inter-arrival times completely, at the expense of being able to give hard- or firm-real-time guarantees.

Lampson proposed a system estimating response time distributions based on runtime information supplied by programs requiring service guarantees.[27] The timing information is presumed to be in form of a distribution. When service is requested, the system can make certain probabilistic guarantees with regard to the service it can provide. However, no algorithm or implementation is presented for such a system.

Atlas, in contrast, predicts execution times based on workload metrics with low error rates. Based on these accurate, per-job execution time predictions,[28] an accurate schedule can be constructed. This allows to give applications not only probabilistic guarantees in form of service-distributions, but per-job feedback on whether a deadline can be met and if not, by how much it will be missed. The error of this feedback only depends on the error of the runtime prediction and can be compensated for by pre-rolling in slack time.

[25] Aloysius K. Mok and Deji Chen. *A Multiframe Model for Real-Time Tasks*. In: *IEEE Transactions on Software Engineering* 23.10 (1997), pp. 635–645.

[26] Sanjoy Baruah et al. *Generalized multiframe tasks*. In: *Real-Time Systems* 17.1 (1999), pp. 5–22.

[27] Butler W. Lampson. *A Scheduling Philosophy for Multiprocessing Systems*. In: *Communications of the ACM* 11.5 (1968), pp. 347–360.

[28] In contrast to per-task distributions in Lampson's proposal.

ATLAS *on Multi-Processor Systems*

In this chapter I describe how I extended ATLAS to take full advantage of multi-processor systems. ATLAS' multi-processor support spans two main aspects, (1) scheduling serial queues on multiple cores to increase system capacity and (2) support for concurrent queues.

MULTI-PROCESSOR SUPPORT IN ATLAS is a hybrid scheduling scheme, based on partitioned data structures, extended with support for process migration. The implementation of the uni-processor scheduling algorithm presented in chapter *ATLAS on Uni-Processor Systems* can be run on multiple cores if applications, and especially ATLAS runtime worker threads, were statically partitioned among available cores.[1] To meet the goal of ATLAS, to hand developers a powerful and yet easy-to-use API to specify soft-real-time work, ATLAS has to handle load distribution among multiple cores automatically.

[1] Using `sched_setaffinity`.

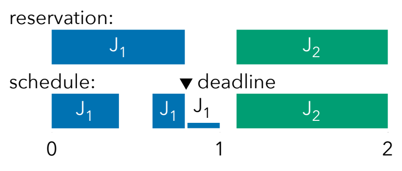
Load Metrics in ATLAS

Effective load-balancing requires a way to measure load first. One drawback of the flexible ATLAS task model is that future load is unknown instead of being neatly summarized in a task utilization, as it is for the Liu and Layland task model. Thus, the load ATLAS has to handle changes as jobs arrive and finish. To determine the load of a run queue I introduce two new metrics. The first metric is the amount of slack time of the first ATLAS job of each run queue. The second metric is based on execution time reservations of jobs submitted so far.

THE SLACK TIME OF THE FIRST JOB ON A RUN QUEUE reflects the immediate load of the run queue and, if the slack is negative, indicates overload and job migration has to be considered.

For job migration to make sense, negative slack of the first job is only a necessary condition for migration. Consider the case when there a single task, implemented by a serial queue, on a run queue, which happens to have negative slack. In this scenario there is no point in migrating the serial queue, since jobs of a serial queue cannot be processed concurrently. Another example, depicted in Figure 19, is when there is a gap between the deadline of the very first job J_1 and the start of the next job, J_2. If this gap is larger than the absolute of the slack time of J_1, it is not necessary to migrate any job, either. Since J_1 has a negative slack it will miss its deadline and migrating the job will only exacerbate the situation by introducing additional overhead. Migrating J_2 has no point either, because its execution experiences no interference from the deadline miss of J_1. The slack of J_2 allows J_1 to finish before J_2 starts executing with elevated privilege, so J_1's completion in ATLAS recovery is not hindered by the execution of J_2.

Figure 19: Migration does not solve all overload situations.



ATLAS TRADES KNOWLEDGE of future jobs for more flexibility in its task model. Contrary to the more traditional Liu and Layland task model, the ATLAS task model has a time-variant utilization. It is paramount to extract as much information as possible from ATLAS' limited knowledge of future execution time requirements.

Any notion of load in the ATLAS task model would, similar to the definition of utilization of an ATLAS task in chapter ATLAS *on Uni-Processor Systems*, depend on the currently available information. A job's contribution to the currently known load, and hence the load of the corresponding task and run queue, changes constantly during processing of a job, because jobs are not periodically recurring.

Let $e_{i,j}(t)$ describe the outstanding execution time of job $J_{i,j}$ at time t, i.e. the execution time that was reserved for $J_{i,j}$, but not yet received by $J_{i,j}$ at time t. Before a job is submitted, i.e. the job is currently unknown to the system, its remaining execution time is zero. At the time of submission, the remaining execution time is set equal to the job's execution time $e_{i,j}$. Between the submission and the release time of $J_{i,j}$ the remaining execution time $e_{i,j}(t)$ equals the execution time requirement $e_{i,j}$. From the release time on, the remaining execution time decreases, as the job receives CPU time.

If, at the deadline of a job, its remaining execution time is not zero, then the job blocked during execution or the system is overloaded. Blocking time does not count as received execution time, and therefore jobs in EDF recovery have a non-zero remaining execution time. Jobs, which missed their deadlines despite having received their execution time reservation, have by definition a remaining execution time of zero.

Formally, the remaining execution time is defined as:

$$e_{i,j}(t) = \begin{cases} 0 & t < r_{i,j} \\ e_{i,j} & t = r_{i,j} \\ e_{i,j} - \text{(received CPU time)} & t > r_{i,j} \end{cases}$$

The definition of the remaining execution time of jobs can be extended to tasks as follows:

$$e_i(t) = \sum_{J_{i,j} \in \tau_i} e_{i,j}(t)$$

Based on the concept of remaining execution time, I define the *interval load function* to describe the amount of execution time reserved by a job $J_{i,j}$ during the interval of absolute times $[t, u]$:

$$l_{i,j}(t,u) = \begin{cases} e_{i,j}(t) & d_{i,j} < u \\ 0 & \text{otherwise.} \end{cases}$$

The interval load function extends naturally to tasks and run queues. Let τ_i be a task. The interval load function for task τ_i is defined as the sum of the interval load functions of all jobs over this task, namely $l_i(t,u) = \sum_{J_{i,j} \in \tau_i} l_{i,j}(t,u)$. The interval load function of the run queue of processor P_k is defined, in turn, as the sum of the load function over all tasks τ_i on the run queue of processor P_k: $l_{P_k}(t,u) = \sum_{\tau_i \in P_k} l_i(t,u)$.

The interval load function is not an exact measure, but merely an approximation of load. A task might have jobs which overran their execution time reservation, but the CPU time required to complete those jobs is not accounted for.[2] Another example where the interval load function does not calculate the load accurately is the case when there is a job with large execution time and deadline just beyond the interval in question.[3] The interval load might therefore be an *underestimation*. On the other hand, jobs might require less execution time than predicted. Hence the interval load might be an *overestimation*.

[2] Without a clairvoyant component the additional execution time required on a deadline miss cannot be determined.

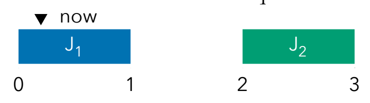
[3] Of course, the interval load function can be adapted to handle that case and account for partial jobs. I found this not to be necessary. Partial execution time can be accounted by splitting the task in the LRT schedule or by considering pre-roll. Which yields better results is out of the scope of this thesis. The technique of partial accounting of remaining execution time can also be used for task splitting or the introduction of virtual deadlines to approximate the fluid schedule of a task. Ion Stoica and Hussein Abdel-Wahab. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Tech. rep. TR-95-22. 1995.

The interval load function is not an analytical tool, but a metric used to compare run queue Atlas-load at runtime. Imagine a job $J_{i,j}$ being submitted at time t. To select a run queue for processing $J_{i,j}$, the interval load function can be computed for all run queues with the interval $[t, d_{i,j}]$ as argument. The capacity of a run queue k is then given by $c_{P_k}(t,u) = l_{P_k}(t,u) - (u - t)$. The run queue capacity can be used to select a suitable run queue for $J_{i,j}$ according to a placement policy. Furthermore, the capacity metric gives an indication of system overload, namely when no run queue has enough capacity to execute $J_{i,j}$.

Although limited in knowledge and approximate, the interval load function allows a look further into the future than the slack-time based metric. I use the load metric of the interval load function for thread migration.

Both presented load metrics are not equivalent. Consider the case in Figure 20, where the first job J_1 has a negative slack time. After a large gap a second job, J_2, follows. Clearly, the slack-time metric indicates an overload situation. The interval load function will only indicate overload for $1 \le t < 1 + |s_{\mathrm{J}_1}(t)|$. If, for example, the deadline of the last job is selected as t, the interval load-method will not indicate overload, since the total time is larger than the required execution time of both jobs J_1 and J_2.

Figure 20: The slack time metric and interval load-metric are not equivalent.



The inverse of the case in Figure 20 cannot happen. When ever the interval load-method will indicate overload, so will the slack-time-method.

To use the interval load function to compare load across run queues, a suitable interval $[t,u]$ has to be used. In the implementation of Atlas I use the current time as the start of the interval, t. For the end of the interval, u, there are number of choices, for example the maximum deadline, the minimum maximum deadline,[4] or a fixed time. Another choice is whether minimal/maximal operations should be performed globally or per-run queue. I will discuss the choice of interval whenever I employ the interval load function.

[4] The minimum of the maximum of each run queue:

$$\min_{0 \le k \le m} \left(\max_{J_{i,j} \in P_k} (d_{i,j}) \right)$$

Load Balancing

After establishing metrics to compare load of run queues, I present the design of load balancing mechanisms in Atlas and how I integrated them in the substrate of the Linux scheduler framework.

ATLAS has two scheduling modes, *consolidate-to-idle* and *race-to-idle*. Consolidate-to-idle[5] aims to minimize the number of active cores, while race-to-idle spreads the work to all available cores.

[5] Marcus Völp, Johannes Steinmetz, and Marcus Hähnel. *Consolidate-to-Idle*. In: *19th Real-Time and Embedded Technology and Applications Symposium*. Vol. 19. Work-in-Progress Proceedings. IEEE. 2013, pp. 9–12.

CONSOLIDATE-TO-IDLE tries to minimize the number of CPU cores used to schedule a task system while still fulfilling the execution time reservations for all jobs. Using a minimal number of CPU cores aims to maximize energy efficiency by letting inactive cores enter a power conserving mode under the operating system's control.[6] Energy efficiency is crucial for mobile applications to extend the battery life of the mobile device.[7] However, a CPU not used by ATLAS may still be used by CFS to accomplish work. ATLAS does not employ DVFS or powers off cores itself. ATLAS consolidates jobs to fewer cores, leaving more CPU time on the freed cores to lower scheduling layers, such as the idle layer. At that point, the operating system's native power management can put an idle core in a sleep state or reduce its core frequency.

[6] Whether consolidate-to-idle or race-to-idle achieves minimal energy consumption for a given workload is system-dependent. Connor Imes et al. *POET: A Portable Approach to Minimizing Energy Under Soft Real-Time Constraints*. In: *Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2015, pp. 75–86.

[7] Etienne Le Sueur and Gernot Heiser. *Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns*. In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. USENIX Association. 2010, pp. 1–8. Etienne Le Sueur and Gernot Heiser. *Slow Down or Sleep, that is the Question*. In: *USENIX Annual Technical Conference*. USENIX Association. 2011.

An optimal solution to the minimum number of cores is NP-hard in the strong sense, since it is reducible to the bin-packing problem. Algorithms which find optimal solutions to the bin-packing problem in a small amount of time are known. Optimal algorithms are too slow to be used in an online scheduler. Approximation algorithms like Best-Fit-Decreasing (BFD) and First-Fit-Decreasing (FFD) are significantly faster and, for a small number of elements, are accurate, often finding an optimal solution.[8]

[8] Korf, "A New Algorithm for Optimal Bin Packing".

However, FFD and BFD require sorting the elements by size, which is not possible in the case of ATLAS-MP. ATLAS jobs are already sorted by increasing deadline and blindly assigning jobs to cores in any order would certainly cause some jobs to miss their deadlines. Delivering real-time guarantees is the primary goal of ATLAS and takes precedence over the secondary goal of minimizing the active core count. Hence, the order of jobs, as determined by ATLAS, has to be preserved. To conserve the order of jobs determined by ATLAS, I will consider the Best-Fit algorithm to attain the placement goal of minimal core count.

RACE-TO-IDLE is characterized by maximal parallel execution, scheduled as early as possible. While ATLAS-MP can spread its work over all available cores, ATLAS-MP uses LRT to schedule execution of jobs. LRT schedules jobs as late as possible. On the other hand, ATLAS' pre-rolling can be considered executing jobs as early as possible.

I use the Worst-Fit heuristic to assign jobs and processes to CPUs with the goal of dividing execution time as evenly as possible between CPUs.

THE LINUX SCHEDULING FRAMEWORK itself only supports thread placement upon thread creation and unblocking. A newly created thread never has ATLAS priority, so placement is delegated to CFS.[9]

[9] Wächtler, "Look-Ahead Scheduling", p. 50

Migrating a thread after it unblocks has potentially lower associated cost than migrating a running thread, because the cache working set of a running thread is hot, in contrast to a blocked thread. Whether ATLAS performs load balancing when an ATLAS-thread unblocks is configurable by `sysctl`.

In race-to-idle-mode ATLAS migrates a thread away from the current run queue to the lowest loaded run queue in the system only if the current run queue is overloaded.

Whole thread migration has the drawback that a thread might require a lot of computation time and does not fit on any run queue without overloading the run queue. A more fine grained method of migration and load balancing is required. For this reason, ATLAS is complemented with job-based migration which allows ATLAS to process different jobs of the same thread on multiple CPUs. Job-based migration differs substantially between concurrent and serial work queues. For this reason, I discuss migration for both queue-types separately in the next two sections.

Serial Queues

Job-based migration acts only on the first job of every thread. A job migration has three steps. First, a job is removed from a run queue. Second, the corresponding thread is migrated to the target run queue.[10] Third and finally, the job is sorted into the new run queue. The thread is now ready to be scheduled on the new CPU and process the migrated job.

[10] I copied the required sequence of calls to `deactivate_task`, `set_task_cpu`, and `activate_task` almost verbatim from CFS.

Migration cannot be performed on just any thread. For a thread to be eligible for migration a technical, a logical and an organizational precondition have to be met:

Technical: Linux requires that a task must not be running when it is migrated.

Logical: The affinity mask of a thread must allow for it to be migrated to the new CPU.

Organizational: To simplify migration and scheduling, a thread can only be migrated, if all its jobs are currently on a single run queue. This avoids spreading a task's jobs to more than two run queues. This also requires that a thread is marked once a job is migrated.[11]

[11] Spreading jobs to more than two run queues does not make much sense, since jobs cannot be processed concurrently.

When a job is migrated, ATLAS must not process the remaining jobs in the old run queue. The same mark used to prevent spreading of jobs of a single thread to more than two run queues is used to determine whether a thread is currently migrated away. Such jobs are ignored by ATLAS in scheduling decisions in the same way ATLAS skips jobs of blocked threads.

An exceptional situation arises when the thread of a job selected for migration has unfinished jobs that missed their deadline. Because those deadlines already have been missed they are ordered strictly before the deadline of the next real-time job. For the next real-time job to be started, all other jobs in front of that job need to finish first. For that reason, unfinished jobs that missed their deadline are migrated along with the ATLAS job.

JOB MIGRATION COMES IN TWO FLAVORS, depending on how it is invoked. An overloaded run queue *pushes* jobs away, while an idle run queue *pulls* jobs towards itself. Both, overload-pushing and idle-pulling, can be separately enabled or disabled using `sysctl`.

If a run queue is idle, i.e. there are no real-time or non-real-time ATLAS jobs queued, idle-balancing causes a job to be migrated from another run queue. To find a suitable job, the run queues are sorted by descending load and searched for a job ready for migration. A job is ready for migration, if its thread fulfills all preconditions for migration. Currently, only a single thread is migrated.

Initially, I used the condition $l_{P_k}(t,u) > u - t$, where $t = \text{now}$ and $u = \max_{P_k} d_{i,j}$ to initiate overload condition. Less formally, a run queue is overloaded if the job with the latest deadline on the run queue will miss its deadline, because the total execution time required by all preceding jobs is larger than the available time from the current point in time to that deadline. Measurements showed that the insensitivity of the interval load function to some overload situations, mentioned in section *Load Metrics in* ATLAS, occurs rather often. In these situations, at least the first job has negative slack, but there is enough time for currently last job to meet its deadline. Thus, the interval load function does not indicate overload, when migration would reduce the total number of deadline misses.

To remedy this situation, I changed overload detection to the slack time method. Whenever the first job on the run queue has negative slack, overload pushing is initiated. While this method performs significantly better, I elaborate in section *Improvements* on additional points of optimization for the new overload detection mechanism.

To avoid any additional overhead on the already overloaded run queue, overload pushing is not carried out actively by the overloaded run queue, but passively by informing other run queues of the overload situation. A run queue notified of overload on another run queue runs the work-stealing algorithm idle-pulling described above, contingent on itself not being overloaded. Other CPUs are notified by issuing an IPI[12] to them.

[12] Inter processor interrupt.

Sending IPIs in an overload situation can be implemented in two ways. A run queue can send IPIs to other CPUs all at once or chained, one after another. In chained notification, the overloaded run queue notifies just one other run queue of the overload situation. If the notified run queue could not resolve the overload situation, it notifies the next CPU, which then tries to resolve the overload situation. The chain ends when either the overload situation could be resolved or all CPUs have been notified. The trade-off is as follows: Notifying all CPUs at once may potentially cause contention on the run queue lock of the overloaded run queue or cause unnecessary IPIs when the first notified run queue resolves the overload situation.

Chained notification, on the other hand, has a potentially higher delay until the overload is mitigated. I chose to notify all run queues at once to minimize delay. Furthermore, the latency of delivering an IPI is highly variable, self-mitigating any contention issues on the run queue lock.

Once a migrated job has been finished – the `next` system call has been invoked – the thread can be migrated back. But this is not the only option. Another idea is to migrate the next job to the current run queue to be processed there. This avoids, or at least holds off on, migrating the thread back, amortizing migration cost over more jobs. Migrating a job is a relatively cheap operation, compared to migrating a thread. This optimization is only used if the next job would not overload the current run queue, otherwise the thread is migrated back. The thread is also migrated back if there are currently no more jobs submitted for this thread.

As long as a thread has Atlas jobs queued, be it real-time jobs, jobs in EDF Recovery, or jobs in CFS Recovery, that thread may not be migrated by CFS. To prevent threads from migrating in other scheduling classes while they have Atlas jobs queued, the affinity mask of such threads is modified by Atlas to only allow them to run on the current CPU.

There are two cases in which an Atlas thread might be scheduled by CFS (1) during pre-roll and (2) if a job is in CFS Recovery. The issue with threads being migrated in scheduling classes other than Atlas is that Atlas is not notified of this migration. If migration in foreign scheduling classes were allowed, there are two possibilities to handle the situation once Atlas notices the migration. The first solution is to simply migrate the thread back. The second solution would be to leave the thread and instead move all Atlas jobs to the new run queue. Let me discuss why I instead chose to prevent migration in other scheduling classes altogether.

Migrating a thread incurs additional cost for migrating the thread back and complicates the implementation of Atlas. Migration always requires taking two run queue locks, for the source and destination run queue. To avoid deadlocks, the run queue locks have to be taken in proper order. Since large parts of the scheduling framework are invoked with the run queue lock already held, this would require potentially dropping the run queue lock and reacquiring it, creating the possibility to violate the invariant protected by the run queue lock.[13] Locks are used to protect data which is temporarily in an inconsistent state. Whenever the lock is released the data must be in a consistent state. The problem with dropping locks, that where acquired by the caller of the current function, is that the called function does not know which modifications the caller made and hence which inconsistencies might exist. Even if we assume the called function had that knowledge the problems continue. Often locks are taken under some preconditions. Those are checked by the caller who took the lock and are not known by the called function who dropped the lock and now needs to re-acquire it. Even if such a knowledge is available it would

[13] After all, the Linux kernel code is hardly commented. Such an endeavour might just open Pandora's box.

have to be maintained for each caller. After all in the Linux kernel almost any function can call any other function – recursively. Another issue is that any local variables that were assigned values depending on data protected by locks should be considered invalid, after the lock has been dropped. Even worse, the entire code path taken when the lock was held might not be valid anymore, when the lock is re-acquired.[14]

[14] Andrew D. Birrell. *An Introduction to Programming with Threads*. Tech. rep. 35. DEC Systems Research Center, Jan. 1989.

The second option to handle thread migrations by CFS is to move jobs to the new run queue of the thread. This is not a good solution either, mostly because this would cause run queue load to be moved in unpredictable ways by scheduling classes that are unaware of the real-time load that Atlas run queues carry.

Preventing migration in scheduling classes other than Atlas seems the most practical solution to the problem.

Concurrent Queues

A serial queue, backed by a single worker thread, is the Atlas correspondent to a task in real-time scheduling theory. While load balancing of serial queues allows for job-level migration, it does not allow for intra-task parallelism. So far, Atlas-MP is able to load balance a single application with multiple serial queues, or multiple applications with one or more serial queues each.

Concurrent queues in GCD allow for independent jobs without precedence constraints to be processed truly concurrently. In this section I present the design and implementation of the Atlas pendant to concurrent GCD queues, allowing applications to take advantage of the inherent parallelism in their soft-real-time jobs. At the same time, concurrent queues afford applications a more light-weight approach to load balancing than serial queues currently do.

Concurrent Atlas queues are based on a thread pool of worker threads, managed by the Atlas runtime. I extended the Atlas system call interface to allow the kernel scheduler to be notified of the creation of thread pools, threads joining and leaving a thread pool, and jobs being submitted to a thread pool.

`atlas_tp_create` sets up management data structures in the kernel and allocates a thread pool ID, which is passed to the user space as result. This call does not create actual threads.

`atlas_tp_join` is called by an Atlas thread pool worker thread to join a thread pool. The thread pool the calling thread wishes to join is identified by the thread pool ID, which is an argument of the system call. Before a thread can join a thread pool it must be pinned to exactly one CPU. Thread pool worker threads should use distinct CPUs to achieve optimal performance. Thread pool worker threads must not migrate, reducing the load balancing problem to migrating thread pool jobs instead of migrating threads and jobs. There is no corresponding system call for a thread to leave a thread pool. Once joined, the only way for a thread to leave a thread pool is to exit.

`atlas_tp_destroy` frees all resources associated with a thread pool. A thread pool can only be destroyed after all threads in a thread pool have quit.

`atlas_tp_submit` queues work for the thread pool to process. Initially, a job is assigned to a thread pool worker thread with a heuristic matching the race-to-idle or consolidate-to-idle load distribution mode.

Scheduling concurrent queues is similar to R-EDF.[15] When a new job is submitted to a concurrent queue, a run queue with enough remaining capacity to accommodate the new job is selected among the run queues with worker threads assigned to the concurrent queue. If multiple run queues have enough capacity, a run queue is selected according to the current scheduling mode and heuristic. In contrast to R-EDF, ATLAS does not keep a running total of the current load on each run queue. Instead, the interval load function is evaluated over the interval from the current point in time until the deadline of the new job.

[15] Sanjoy Baruah and John Carpenter. *Multiprocessor fixed-priority scheduling with restricted interprocessor migrations*. In: *Proceedings of the 15th Euromicro Conference on Real-Time Systems*. IEEE. 2003, pp. 195–202.

In race-to-idle mode the Worst-Fit and Next-Fit heuristics can be used to assign a newly submitted job to worker threads and hence place the job on a run queue. The Worst-Fit heuristic selects always the run queue with the lowest load, which is also well-defined during overload situations.

In consolidate-to-idle mode the Best-Fit and First-Fit heuristics can be selected to place a new job on a worker thread. While this assignment strategy might not result in completely idle run queues, it aims to concentrate the load of ATLAS jobs on as few as possible run queues. Run queues with few ATLAS jobs are relieved of as much load as possible, maximizing their idle time.

If no run queue has enough capacity for the newly submitted job, the job is submitted to the run queue with the lowest load, irrespective of the load-balancing mode. This is to ensure that as few as possible jobs miss their deadlines and the amount of tardiness is not unnecessarily increased.

The ATLAS scheduler needs to adapt the assignment of jobs to worker threads during runtime to compensate for external interference, such as blocking, scheduling overhead, and higher scheduling classes requiring CPU time. Pinning worker threads in thread pools reduces the cost of load balancing compared to serial queues. Since worker threads already exist on different CPUs, threads need not be migrated. It is sufficient to migrate only jobs, which is a comparatively cheap operation.

Concurrent queues take part in overload-push and idle-pull job migration. There is only a single precondition for a job of a concurrent queue to be eligible for migration: the job must not have been started. In contrast to serial queues, where a thread is migrated across CPUs, including its current execution state, migrating a job leaves the execution states of all worker threads on their CPUs. Once a thread started processing a job, that job cannot be migrated.

Job migration is a four step process.

(1) The job has to be removed from the local run queue and the current thread pool worker it is assigned to.

(2) The `task_struct` of the worker thread on the destination run queue has to be found. This information is maintained in the thread pool data structures in the kernel.

(3) Once the new thread is known, the job is inserted into the new thread's job list.

(4) Finally, the job is enqueued in the new run queue.

GCD processes jobs in submission order. As long as deadlines of successive jobs are monotonically increasing, ATLAS conforms to the GCD FIFO-behaviour. A problem arises, when a later submitted job has an earlier deadline than the currently latest deadline. There are four ways to handle this situation (1) reject the job, (2) adjust the deadline of the new job, (3) adjust the deadline(s) of older job(s), and (4) re-order jobs.

Forcing monotonically increasing deadlines on the programmer is not very appealing, especially if there are three choices of meaningful semantics for this scenario to choose from.

Adjusting the deadline of the new job is problematic, because an adjustment to make the deadlines monotonically increasing would mean to *add* to the deadline. As a result, the job could potentially finish later than the programmer intended to – not a very intuitive solution.

Adjusting the deadlines of already submitted jobs would mean to push their deadlines before the new deadline. While this would not cause jobs to finish later than their initial deadline required, there are two problems with this solution. For one, the slack for jobs, which ATLAS uses for interactivity, would unnecessarily be reduced. Second, and probably even worse, reducing deadlines of earlier jobs could lead to artificially created overload situations and deadline misses.

The fourth option keeps the deadlines of jobs as they are, but re-orders jobs to maintain the invariant of monotonically increasing deadlines. The issue with this solution is that the user space needs to be informed, when job re-ordering takes place. For this reason, each job is assigned an ID by user space and passed to kernel space with the `submit` system call. I extended the `next` system call with an out-parameter which is used by the kernel to inform the user space of next job to be processed. With this change to the `next` system call the FIFO-processing assumption between kernel and user space is broken up and the kernel space is free to re-arrange jobs. I back-ported this feature with the same semantics to serial ATLAS queues.

Utilization Bounds of ATLAS-MP

In this section I discuss the expected utilization bounds of ATLAS, if ATLAS is used to schedule periodic task systems, specifically task systems adhering to the Liu and Layland task model.

According to the categorization of real-time multi-processor scheduling algorithms proposed by Carpenter et al.,[16] ATLAS serial and concurrent queues have different theoretical utilization bounds.

[16] Carpenter et al., "A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms".

ATLAS SERIAL QUEUES are in the class of (2, 3)-restricted scheduling algorithms, meaning serial queues have job-level dynamic priority assignments and offer unrestricted migration. Migration can occur at any instant, but the priority of threads changes only on job boundaries. The theoretical utilization bound listed by Carpenter et al. is

$$\frac{m^2}{2m-1} \leq u \leq \frac{m+1}{2}.$$

The lower bound of this class of scheduling algorithms is achieved by algorithm EDF-US[$m/2m-1$],[17] while the upper bound is achieved by algorithm fpEDF,[18] an improved version of EDF-US. Both scheduling algorithms are based on giving the highest priority to "heavy tasks", that is tasks with a utilization above a threshold ς. While EDF-US uses a threshold of $m/2m-1$, fpEDF uses a threshold of 0.5. All tasks with utilization below the threshold are assigned priorities according to EDF.

[17] Srinivasan and Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors"

[18] Baruah, "Optimal Utilization Bounds for the Fixed-priority Scheduling of Periodic Task Systems on Identical Multiprocessors".

Since ATLAS builds its schedule strictly by deadline, the achievable utilization bound is, depending on job and task utilization, arbitrarily close to 1. ATLAS suffers from Dhall's effect. Goossens, Funk, and Baruah have given a utilization bound based on the maximum per-task utilization u_{max}.[19] This bound is

[19] Joël Goossens, Shelby Funk, and Sanjoy Baruah. *Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors.* In: *Real-time systems* 25.2-3 (2003), pp. 187–205.

$$u = m(1 - u_{max}) + u_{max}.$$

This is the best bound ATLAS serial queues are currently able to achieve. If ATLAS were to prioritize tasks and jobs with utilization above a threshold ς, like EDF-US and fpEDF do, ATLAS could achieve a better utilization bound for serial queues.

CONCURRENT QUEUES are in the class of (2, 2)-restricted scheduling algorithms. Once started, a job must not migrate to another CPU. The same as with serial queues, tasks change priority only at job boundaries. The theoretical utilization bounds for (2, 2)-restricted scheduling algorithms are

$$m - \alpha(m-1) \leq u \leq \frac{m+1}{2},$$

where α is a real number, such that $\alpha \leq u_i, \forall \tau_i \in \tau$.

The utilization bound of ATLAS concurrent queues is currently limit by two factors. ATLAS concurrent queues (1) lack preemptibility and (2) do not prioritize 'heavy' jobs.

Preemptivity of jobs is necessary for priority-driven scheduling and a key requirement to feasibly schedule task sets with a utilization equal to machine capacity, even on uni-processor machines.

Without considering a job's utilization, ATLAS concurrent queues are only able to achieve the lower utilization bound, as R-EDF does.[20]

[20] If they were not already limited by non-preemptibility.

For task systems where the maximal utilization of each task is 0.5, the lower and upper bound coincide. For task systems with lower maximal task utilization than 0.5, the lower bound will exceed the upper utilization bound. To achieve the upper utilization bound, tasks with utilization higher then 0.5 need to be prioritized over any other tasks in the system. For this purpose the R-PRID scheduling algorithm has been proposed.[21]

[21] Baruah and Carpenter, "Multiprocessor fixed-priority scheduling with restricted interprocessor migrations".

It is currently unknown how the scheduling classes (2, 2) and (2, 3) compare.

The utilization bounds discussed above are only valid for each queue type used in isolation. For concurrent queues, m is the number of worker threads, which must not exceed the number of available processors. More than one concurrent queue can be used at the same time, but the analysis has to be performed for each queue in isolation. Processor assignments have to be exclusive to exactly one queue.

Improvements

ATLAS-MP is a research prototype and as such does not yet incorporate a number of optimizations. Instead, I focused on correctness, flexibility and a clean implementation. This section includes known deficiencies and optimization opportunities which could not be implemented in the prototype due to time constraints. Optimizations applicable to ATLAS-MP include:

SEVERAL PARAMETERS, such as the number and the total capacity of jobs submitted to a task or queued on a run queue, are calculated on demand. Another parameter is whether a thread has a migrated job or not. Clearly, these computations have linear complexity in the number of jobs and require traversal of pointer based data structures, increasing the probability of cache misses. Both properties may have a negative impact on performance. These parameters need not be calculated on demand. They can be implemented as a running total which has constant complexity for updating and reading the value. A running total is a worthwhile optimization especially for values updated seldom.

JOB-STEALING IS ANOTHER POINT for optimization. Currently only a single job[22] of a single thread is migrated. Since migration has a high associated cost in terms of locking, it might prove beneficial to migrate more than a single job from a single thread. Here the design space is rather large, ranging from migrating multiple jobs of a single thread, to a fixed number of jobs from arbitrary threads, or migrating one or more whole threads. Another approach would be to use migration to distribute the load as equally as possible between run queues.

[22] Plus all previous jobs in EDF Recovery and CFS Recovery.

OVERLOAD PUSHING is vulnerable to the case of false-positive signalling of overload indicated in section *Load Metrics in* ATLAS: when a single job queued on a run queue falls behind, overload pushing is initiated, but there are no other jobs which could be migrated away to lighten the load on the overloaded run queue. The amount of negative slack is also not taken into account. The `ktime_t` data type, used in the Linux kernel, handles time in nanosecond resolution. Introducing IPI-overhead in the order of tens of microseconds is unreasonable to resolve "overload" in nanosecond or even microsecond range. A threshold can be used to activate overload pushing only for significant overload situations. On the other hand, such a threshold is hardly machine independent and thus introduces a parameter which has to be tuned for optimal performance.

Evaluation

In this chapter I give a first characterization of Atlas in terms of introduced overhead, schedulability, and functionality. All experiments have been run on a quad-core Intel® Core™ i7 CPU, model 860 running with 2.8 GHz and 4 GiB of RAM. I disabled hyper-threading for all experiments.

System Call Overhead

Figure 21 shows the overhead of the uni-core Atlas `submit` and `next` system calls, depending on the number of queued jobs. Values smaller than the first or larger than the third quartile are drawn in light color. Values between the the first and third quartile are drawn in dark color. The median is accentuated as a white line. I obtained the data by repeating each experiment 10,000 times and discarding the top percentile of measurements as outliers.

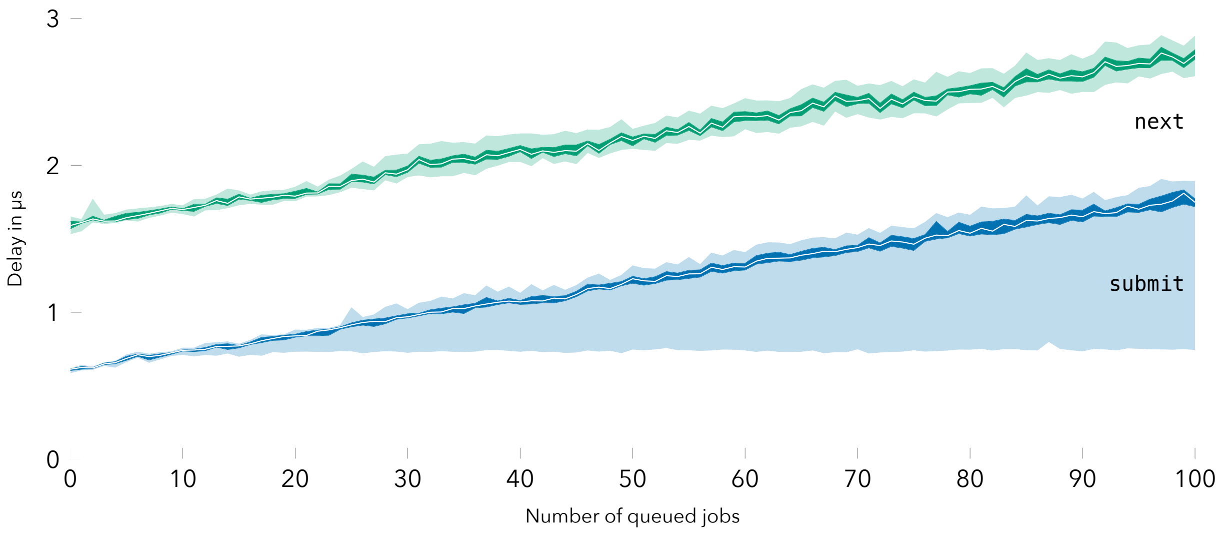


Figure 21: Latency of the Atlas `submit` and `next` system calls, depending on the number of already-queued jobs.

The overhead of both system calls is acceptable, with a median of 0.9 µs for the submit system call and 1.8 µs for the next system call, both with 25 queued jobs. I expect applications to queue no more than two dozen or so jobs at any time. The FFplay demo[1] application queues at most 20 jobs. With a combined overhead of 2.7 µs per job, this benchmark also shows that Atlas is apt to handle fine-grained work.

[1] Roitzsch, "Practical Real-Time with Look-Ahead Scheduling"

In the rest of this chapter I discuss the evaluation of schedulability experiments. I loaded Atlas with generated task sets to measure its utilization bounds. Unfortunately, there seems to be no well-established consensus on how to generate tasks sets for multi-processor real-time systems. Thus, I explain how I generated the task sets first.

Task Set Generation

The task set generation algorithm I devised, UUniMulti, has four input parameters: the number of tasks n, the total utilization of the task set

u_{sum}, the maximum utilization of any task u_{max}, and the minimal and maximal periods of the generated tasks p_{min} and p_{max}.

At first, utilization values for n tasks are generated in such a way, that the following two constraints are satisfied:

$$\sum_{\tau_i \in \tau} u_i = u_{sum} \text{ and } \forall \tau_i \in \tau : 0 < u_i \leq u_{max}$$

My method of generating utilization values is based on *UUniSort*,[2] but less elegant. For each task its minimum and maximum utilization are determined and a random value is chosen from that interval.

[2] Enrico Bini and Giorgio C. Buttazzo. *Measuring the Performance of Schedulability Tests*. In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

To explain the need for a minimum and maximum utilization of a task, let us consider generating a task set for the parameters $n = 3$, $u_{sum} = 1.5$ and $u_{max} = 1$. The first task can have any utilization in the interval $(0, 1]$. Let us assume, utilization 1 was chosen. Since the total utilization of the task set is required to be 1.5, the utilization of the second and third task has to be smaller than 0.5.

The utilization of the second task is chosen from the interval $(0, 0.5 - \epsilon]$. ϵ is the resolution at which utilization values are generated, for example 0.001.[3] Subtracting ϵ for each remaining utilization value to generate ensures that no task has a utilization of 0. A task system with n tasks, which has a task with utilization 0, is in fact a task system with $n - 1$ tasks, hence task utilization must be non-zero.

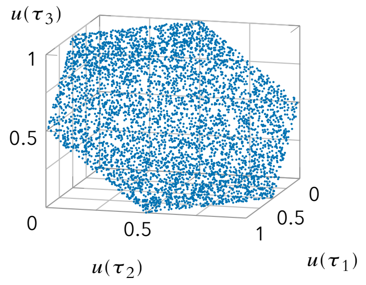
[3] In my implementation I use integer arithmetic. The advantages of multi-precision integer arithmetic over floating point are no rounding errors, higher speed, and arbitrary resolution.

The utilization of the third task is set to be $u_3 = 1.5 - \sum_{i=0}^{n-1} u_i$ to match the desired total utilization of the task set.

To explain the requirement for a minimum utilization let us consider the same task set parameters, but now assume that the first utilization value, drawn from the interval $(0, 1]$, is 0.1. The second utilization value must now be drawn from the interval $[0.4, 1]$, otherwise the third task may have a utilization exceeding 1, violating the Liu and Layland task model.

Figure 22 shows 1000 generated task sets with the parameters of the example task set just considered: $n = 3$ tasks, a maximal task utilization of 1.0 and a task set utilization of 1.5. All points are located on a plane, such that the sum of the three coordinates of each point sum to 1.5. The plane is delimited in each dimension by the allowable per-task utilization of $(0, 1]$.

Figure 22: Distribution of utilization values obtained with UUniMulti for three tasks with a maximum per-task utilization of 1.0 and a task set utilization of 1.5.



THE NEXT STEP is generating either periods or execution times. Execution time generation can be used if a known workload has to be modelled. Execution times can be drawn from a known or derived distribution of execution times. Multiplying the execution times with the utilization value yields the period of the task. The main drawback of this approach is that even for small task systems, of less than ten tasks, the hyper-period grows significantly large, prohibiting actually executing the task system.

The same problem of exploding hyper-periods arises when periods are drawn uniformly from each order of magnitude between the minimum and maximum period lengths, as proposed by Davis, Zabos, and Burns.[4]

[4] Robert Davis, Attila Zabos, and Alan Burns. *Efficient Exact Schedulability Tests for Fixed Priority Real-time Systems*. In: *IEEE Transactions on Computers* 57.9 (2008), pp. 1261–1276.

To bound the hyper-period, I propose to first choose a set of prime factors and then choose an acceptable range of exponents for each prime factor. To draw a period, the exponent for each prime factor is chosen from its respective range. The prime powers are then multiplied to get the period. The maximal hyper-period is the product of highest power of each prime.

The drawback of this method is, that the distribution of the periods is not known. Without any a-priori knowledge of an application, some probability distribution must be assumed.[5] For most experiments a uniform distribution is assumed. My proposed method might contain more harmonic periods than randomly drawn periods. On the other hand, as noted by Davis, Zabos, and Burns, randomly generated periods might not be representative of real applications.

[5] Bini and Buttazzo, "Measuring the Performance of Schedulability Tests".

For my experiments I restrict myself to period lengths between 10 ms and 100 ms as realistic workloads for Atlas. I draw periods from the set $\{4,6,8,9,12\}^2$.

The period is then multiplied by the utilization to calculate the execution time.

Experiments

I used the `strstr` function to emulate work. The work function searches for the string "`test`" in a memory area filled with upper case letters '`A`'. I tuned the size of the memory area such that the invocation of `strstr` takes approximately 100 µs on my test machine. The test function is called repeatedly, to keep the CPU busy for the duration of the generated execution time.

I evaluate serial and concurrent Atlas queues separately. The experiment consists of loading Atlas with a periodic real-time task set for one hyper-period. I repeat this measurement for different core counts, task set utilizations, number of real-time tasks, and Atlas migration settings, to determine the ratio of jobs that missed their deadline. With this setup, I can experimentally determine if Atlas meets the utilization bounds derived in section *Utilization Bounds of* Atlas-MP.

Serial Atlas queues are directly mapped to real-time tasks. For each real-time task a Linux thread is instantiated to process the real-time task's, and hence the serial queues, jobs.

For concurrent Atlas queues all real-time jobs are submitted to an Atlas thread pool with a worker thread pinned to each available CPU. The assignment of jobs to worker threads is left to the Atlas scheduler. I use the Worst-Fit heuristic to assign jobs to worker threads within Atlas concurrent queues.

The Deadline scheduling class is tested with a setup similar to Atlas serial queues. For each real-time task a thread is set up and the Deadline scheduler is fed the parameters of the real-time task for each thread via Linux' `sched_setattr` system call. Because the Deadline scheduling class has no `next` system call on which the worker thread could block after it processed a job, I introduced a queue, into which jobs where submitted. If a thread completed all jobs, emptying the queue, the thread blocks on a condition variable, to be woken up when the next job is released. Running the experiment on a vanilla Linux 4.0 kernel[6] resulted repeatedly in a crash caused by dereferencing an invalid pointer in the Deadline scheduling class code. For this reason, I ran the Deadline scheduling class benchmarks on a vanilla Linux 4.4 kernel,[7] where the particular bug I hit seems to have been fixed.

[6] Git commit `39a8804455fb23f09157341d3ba7db6d7ae6ee76`

[7] Git commit `afd2ff9b7e1b367172f18ba7f693dfb62bdcb2dc`

To measure and compensate for additional overhead for simulating tasks, I fixed the workload at 1 ms, 10 ms, and 100 ms and generated a task set containing a single task with periods 10 ms, 100 ms, 1000 ms and utilization of 0.1. Then I increased the utilization of the task submitted to the kernel until no more deadline misses occurred, while keeping the amount of work done in user space constant. The result of this experiment is the over-allocation that is necessary to compensate any overhead incurred in the kernel or user space.

I found that after an over-allocation of 160 µs no deadline misses occurred for either of the three periods tested for Atlas serial queues. I adapted the workload emulation by subtracting a fixed amount of 200 µs from the execution time. For execution times smaller than 200 µs, no emulation is done and the execution time is taken up solely by kernel and user space overheads. Since the estimation of overhead includes only one thread, the overhead for task switching and/or migration is not accounted for.

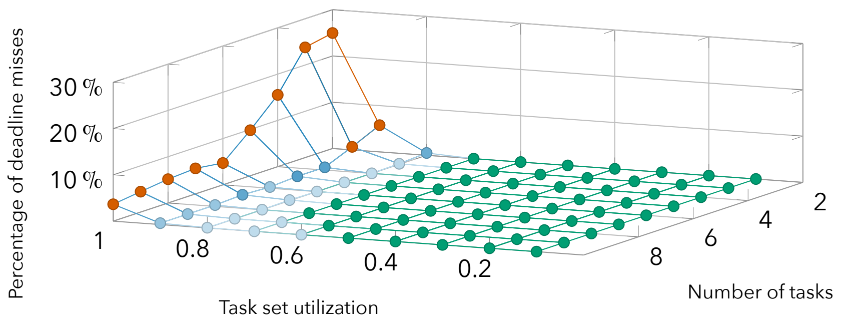
I determined the overhead of the benchmark for concurrent queues to be 250 µs. I rounded the result up to the next hundred microseconds and the set the compensation to 300 µs.

I measured the overhead of the benchmark for the Linux Deadline scheduling class initially at 20 µs and employed an over-allocation of 100 µs to compensate for overhead.

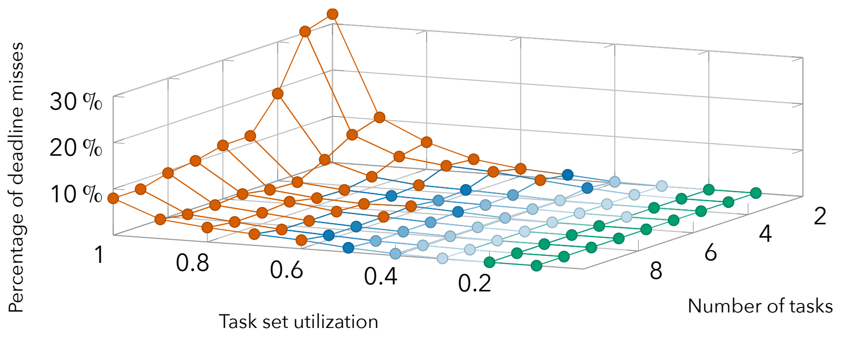
To establish a baseline for any further experiments with multiple cores, I disabled all but one core on my test machine and ran a series of uni-processor tests. I varied the utilization from 0.1 to 1.0, the number of tasks from 2 to 10 and set the maximal per-task utilization to 1.0. For each parameter set I generated 200 task sets and ran them each for one hyper-period on both Atlas queue types and the Deadline scheduling class, counting the total number of missed jobs. The results are plotted in Figure 23.

Data points in green color represent hard-real-time capability; no deadline misses were observed. Data points shaded blue show the soft-real-time band with less than 1 % deadline misses. Orange data points signify a deadline miss ratio higher than 1 %.

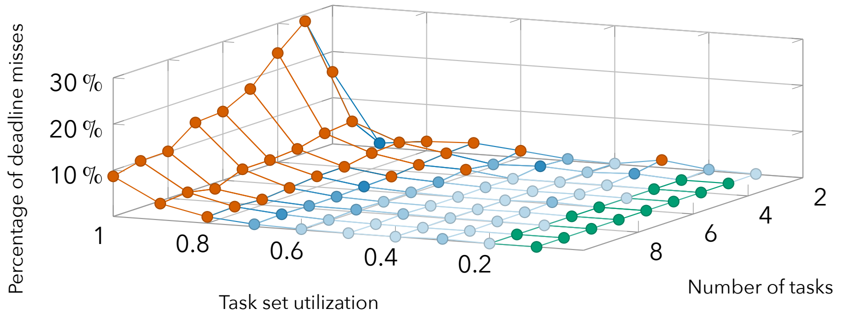
The hard-real-time utilization bound of Atlas serial queues, depicted in Figure 23(a), seems to be at a task set utilization of 0.7. Only for task sets with nine or ten tasks, the hard-real-time utilization bound decreases to 0.6 and 0.5, respectively. Presumably, increased



(a) Atlas serial queue



(b) Atlas concurrent queue



(c) Linux Deadline scheduling class

Figure 23: Achieved utilization bounds of Atlas serial queues, concurrent queues, and the Linux Deadline scheduling class on a uni-processor system.

task switching overhead in the scheduler reduces the utilization bound for these task sets.

More than 99 % of all deadlines are met for task sets with utilization less than or equal to 0.9, except for task sets with 2 or 3 tasks, that reach the 99 % limit for a utilization less than 0.9. The overhead compensation of 200 µs clearly favours task sets with many tasks. More tasks result, on average, in smaller jobs and more jobs, each of which receives the execution time reduction as compensation.

Figure 23(b) shows the result of the same experiment conducted on Atlas concurrent queues.

With concurrent queues, Atlas achieves a significantly lower utilization. Concurrent Atlas queues are not preemptible. Once a worker thread started executing a job, the job must run to completion. Since the number of worker threads equals the number of CPUs, with each worker thread bound to a single CPU, a job might not get executed on time.

I repeated the same experiment for the Linux Deadline scheduling class. Figure 23(c) shows the result.

The Deadline scheduling class has no utilization where for all numbers of tasks all deadlines are met. For most task sets, the Deadline class seems hard-real-time capable up to a utilization of 0.2. From a utilization of 0.8 onward, the Deadline class misses more than 1 % of all deadlines; for task systems with few tasks the utilization may be as low as 0.6. At least for short periods in the range of 10 ms to 100 ms the Deadline scheduling class is inferior to Atlas serial queues.

The utilization bounds of Atlas concurrent queues and the Deadline scheduling class seem similar, leading me to the conclusion, that preemption in the Deadline scheduling class does not work properly. Presumably, the CBS algorithm pushes jobs into the next period.

For the next set of experiments, I enabled two out of the four processors of my test machine. I kept all task set parameters the same, except the task set utilization, which I varied from 0.5 to 2, the capacity of the system.

In Figure 24, the experimentally determined utilization bounds of the Deadline scheduling class are plotted. On a dual-core machine, the best result of the Deadline scheduling class is nearly 20 % of missed deadlines for a task set utilization as low as 0.5. The Deadline scheduling class processing implicit deadline task systems is equivalent of a global EDF scheduler and thus should be able to schedule task systems up to a utilization bound of 1.0.

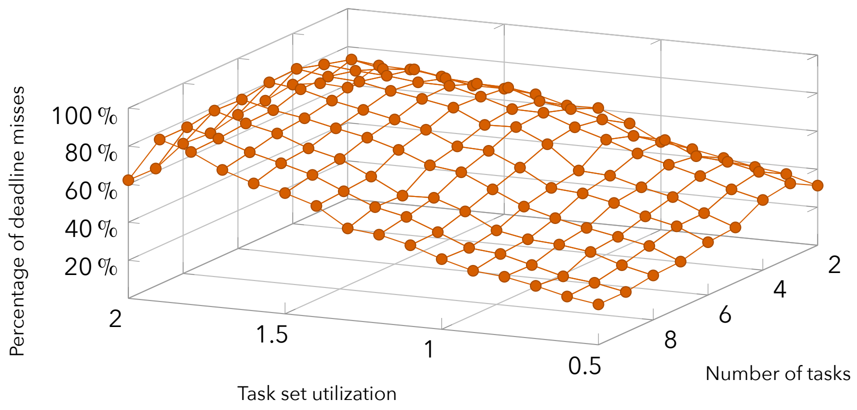


Figure 24: Measured ratio of deadline misses of the Linux Deadline scheduling class on a dual-core system.

Presumably, worker threads overrun their budgets by a small margin. Now, the Constant Bandwidth Server algorithm kicks in and limits the threads to a share equal to their reservation, causing even more deadline misses.

The Deadline scheduling class does not handle overload. If the Deadline scheduling class believes it has no capacity left it will reject a thread changing its priority and the thread will remain scheduled by CFS. Because Atlas does not decline work, the benchmark ignores overload of the Deadline scheduling class and executes the task system either way.

On the left side of Figure 25 the results of Atlas serial queues are shown; the results of Atlas concurrent queues are on the right side. Combinations of Atlas load balancing modes are plotted from top to bottom. In the first row, load balancing is disabled, in the second row idle-pull load balancing is enabled, followed by overload-push load balancing in the third row. The last row shows the effect of both, idle-pull and overload-push load balancing enabled at the same time.

Figure 25(a) shows the utilization bounds of Atlas with load balancing disabled and demonstrates the necessity of load balancing. For experiments where the task set utilization exceeds 1, the deadline miss ratio follows a steep increase, which peaks at approximately 70 %. The only form of load balancing in this set of benchmarks is done by CFS. CFS determines thread placement when a thread is newly created. Only when a thread has completed all Atlas jobs, and hence is about to block, CFS is able to migrate the thread.

Both load balancing modes, shown in Figures 25(c) and 25(e), improve the deadline miss ratio considerably up to a task set utilization of about 1.5. Nevertheless, for task sets with more than utilization 1, more than 1 % of the jobs miss their deadline. The idle-pull load balancing mode performs better than overload-push for task sets with 2 or 3 tasks and utilization values between 1 and 1.5.

If both load balancing modes are combined, the overhead of overload-push dominates the result, which is depicted in Figure 25(g).

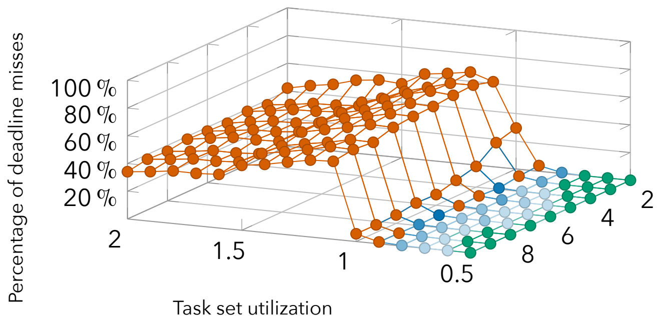
Figure 25(b) shows that the performance of Atlas concurrent queues on two processors is also dominated by the inability to preempt running jobs. Load balancing cannot mitigate this drawback.

Performance of the load balancing option overload-push is shown in Figure 25(d). Because the worker threads are constantly in overload, continuous IPIs cause even more deadline misses.

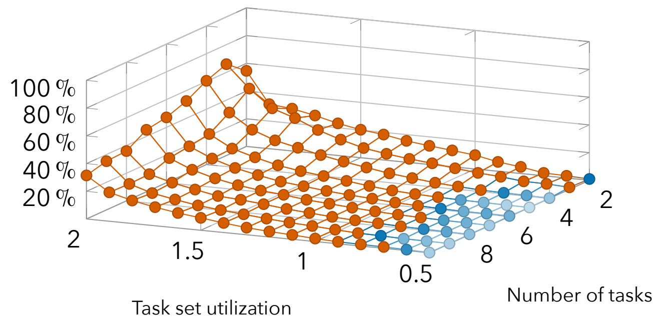
Figure 25(f) is very similar to the performance of Atlas concurrent queues with no load balancing. The reason is that worker threads are rarely idle and thus the idle-pull operation is executed seldom. The influence of idle-pull, which is not visible in the graphic, varies. In some configurations an improvement was measured, whereas in others more deadlines were missed. The minimum was 722 less deadline misses; the maximum 426 more deadline misses of idle-pull against no load balancing. Over all operating points the improvement of idle-pull operation was 3146 less deadline misses. Compared to the 4 781 440 jobs of each experiment in Figure 25, this corresponds to an improvement of 0.066 %. Because of the low number of task sets and jobs, the statistical significance of these is numbers is questionable.

If both load balancing modes, idle-pull and overload-push, are enabled, the deteriorating effect of overload-push dominates the result and no improvement over no load balancing can be achieved.

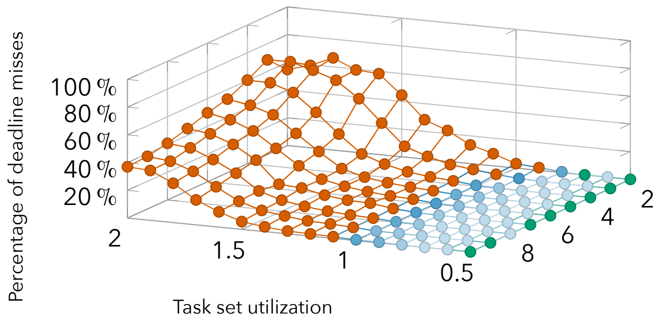
Generally speaking, the more tasks are in the task set, the fewer deadlines are missed. The reason is that the mean task utilization decreases, as task sets with increasing number of tasks are generated for a fixed task set utilization.



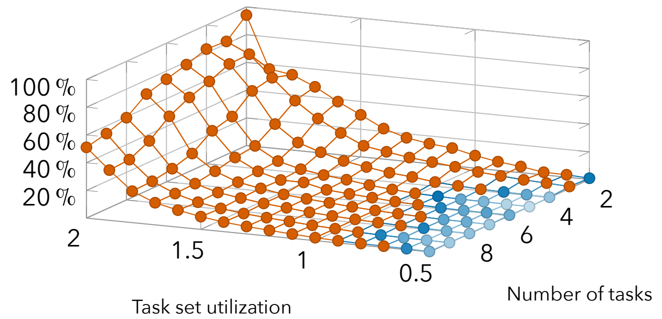
(a) ATLAS serial queue; no load balancing



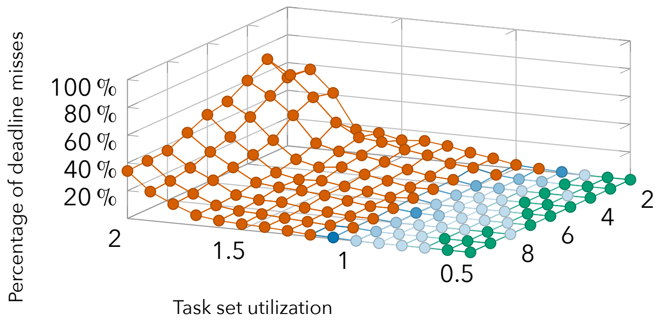
(b) ATLAS concurrent queue; no load balancing



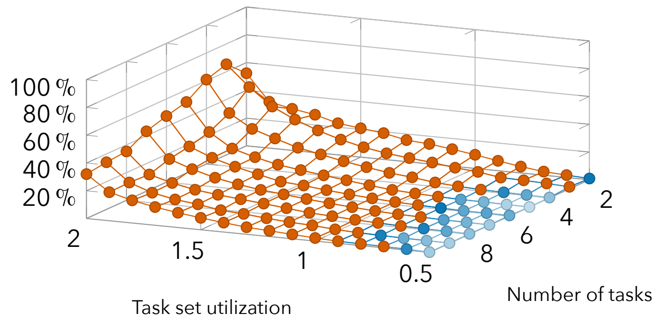
(c) ATLAS serial queue; overload-push load balancing



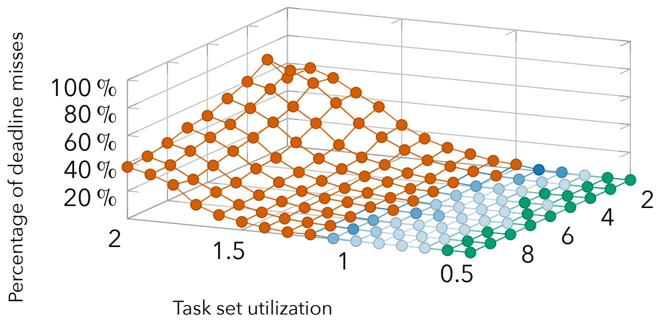
(d) ATLAS concurrent queue; overload-push load balancing



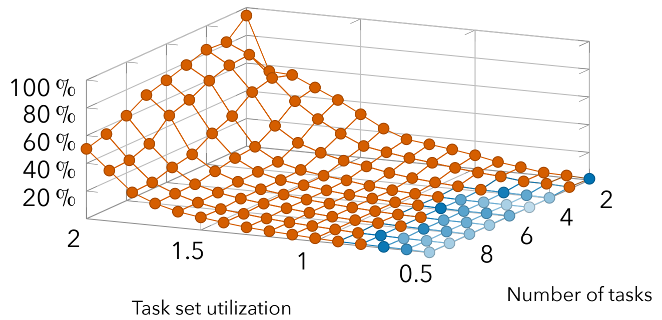
(e) ATLAS serial queue; idle-pull load balancing



(f) ATLAS concurrent queue; idle-pull load balancing



(g) ATLAS serial queue; overload-push and idle-pull load balancing



(h) ATLAS concurrent queue; overload-push and idle-pull load balancing

Figure 25: Achieved utilization bounds of ATLAS serial and concurrent queues with various combinations of load-balancing modi supported by ATLAS on a dual-core machine.

I ran the last set of experiments with all four cores enabled. I varied the task set utilization for all schedulers from 0.5 to 4.0. The rest of the parameters remain the same. The results are presented in Figure 26. Figure 26 has the same composition as Figure 25; the results for serial queues are shown in the left column, those of concurrent queues in the right column. The rows contain, in order, the results for no load balancing, overload-push load balancing, idle-pull load balancing and combined overload-push/idle-pull load balancing.

The results of my measurements on the quad-core system fit well with the performance figures of the dual-core system. Atlas serial queues continue to exhibit less than 1 % of deadline misses for task set utilizations less than 1, as they should.

If no load balancing is enabled, Dhall's effect is clearly visible in Figure 26(a).

Overload-push load balancing, depicted in Figure 25(c), displays moderate deadline miss ratios of less than 10 %, for a task set utilization up to 2 and more than 5 tasks.

Idle-pull load balancing, presented in Figure 25(e), shows similar performance in the region below a task set utilization of 2 and more than 5 tasks like overload-push load balancing. Unlike the dual-core case, where idle-pull performed marginally better, overload-push shows better results in the quad-core system, for this small region. Overall, idle-pull load balancing seems to produce less deadline misses as overload-push load balancing.

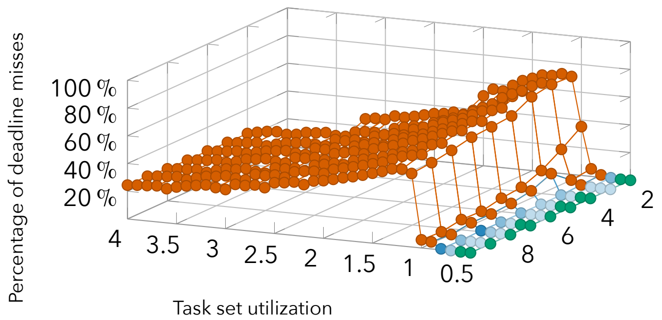
The results for the combination of idle-pull and overload-push load balancing are plotted in Figure 26(g). As before, the data is very similar overload-push load balancing on its own, leading me to conclude the overhead of sending IPIs dominate the effects of idle-pull load balancing. Additionally, with increasing load idle phases become more and more seldom.

The deadline misses for concurrent queues increase on the quad-core system.

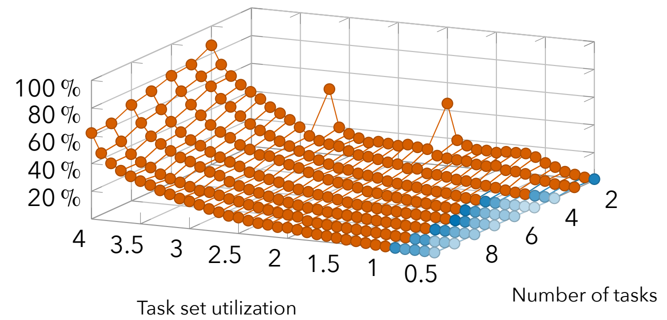
If no load balancing is employed, the deadline miss ratio increases from, for example, around 10 % for 10 tasks at 75 % capacity in the dual-core case to nearly 20 % in the quad-core case, depicted in Figure 26(b).

The performance of overload-push load balancing also worsens, as shown in Figure 26(d). As more CPUs are added to the system, the lack of rate-limiting of IPIs causes nearly all deadlines or near system capacity to be missed. In the dual-core case the deadline miss ratio seemed to plateau around 80 %.

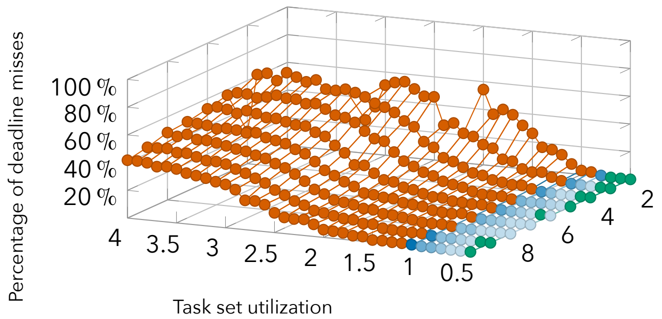
Figure 26(f) shows the results for idle-pull load balancing on four cores. On the quad-core system, the similarity between no load balancing and idle-pull load balancing remains, because idle time remains rare. The overall performance of idle-pull load balancing worsens, going from two to four cores. Picking up the same example as before, 10 tasks at a utilization of 75 % system capacity, the deadline miss ratio in-



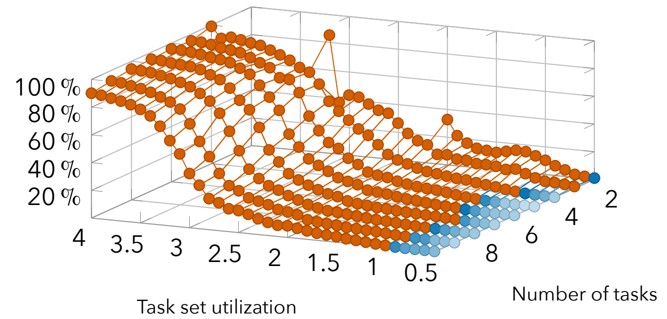
(a) Atlas serial queue; no load balancing



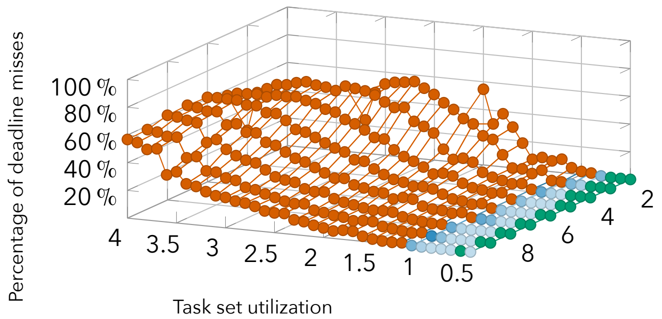
(b) Atlas concurrent queue; no load balancing



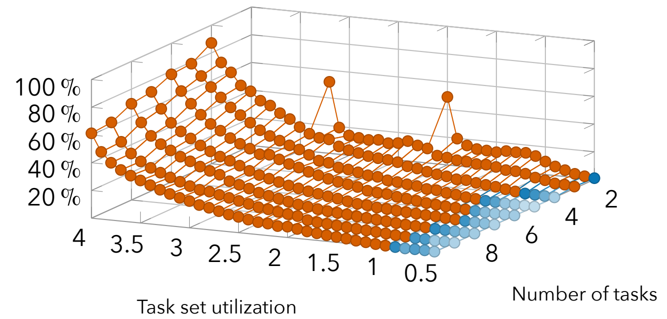
(c) Atlas serial queue; overload-push load balancing



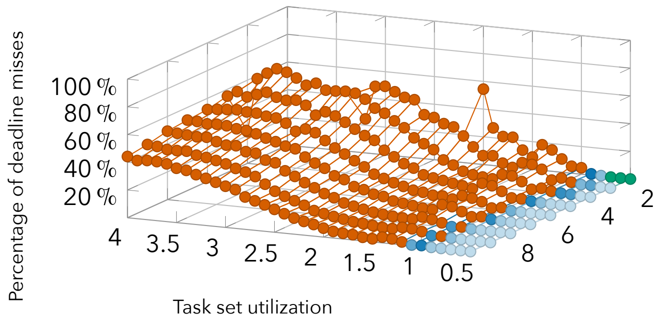
(d) Atlas concurrent queue; overload-push load balancing



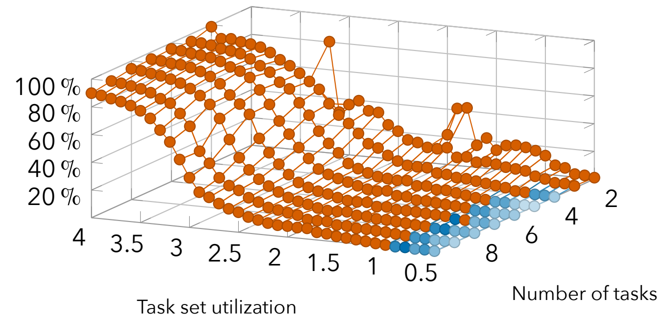
(e) Atlas serial queue; idle-pull load balancing



(f) Atlas concurrent queue; idle-pull load balancing



(g) Atlas serial queue; overload-push and idle-pull load balancing



(h) Atlas concurrent queue; overload-push and idle-pull load balancing

Figure 26: Influence of load balancing on the deadline miss ratio of Atlas serial and concurrent queues on a quad-core machine.

creases from approximately 10 % for the dual-core case in Figure 25(f) to around 20 % for the quad-core case in Figure 26(f).

As before, the performance of overload-push load balancing dominates performance of the combination of overload-push and idle-pull, as depicted in Figure 26(h).

Figure 27 shows the result of the benchmark for the Deadline scheduling class on a quad-core system. As before, on the dual-core system, the Deadline scheduling class is not able to meet an acceptable number of deadlines, even below its theoretical utilization bound of 1.0.

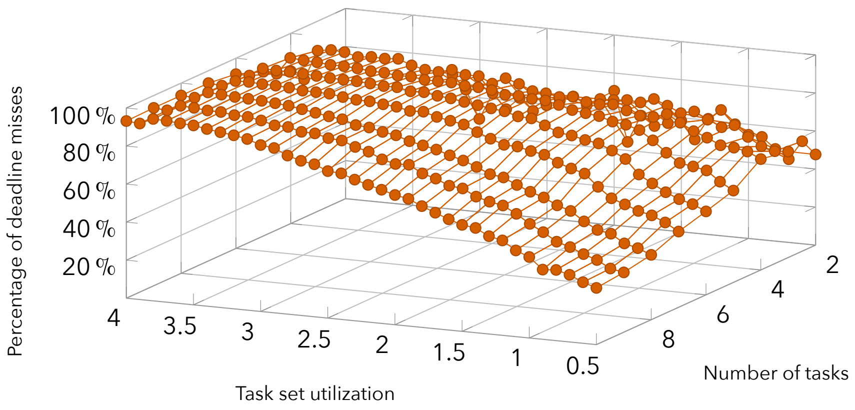
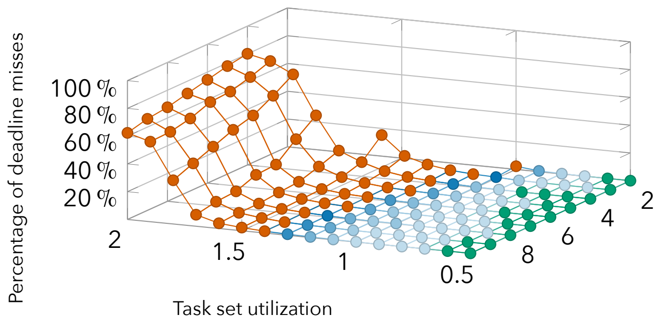


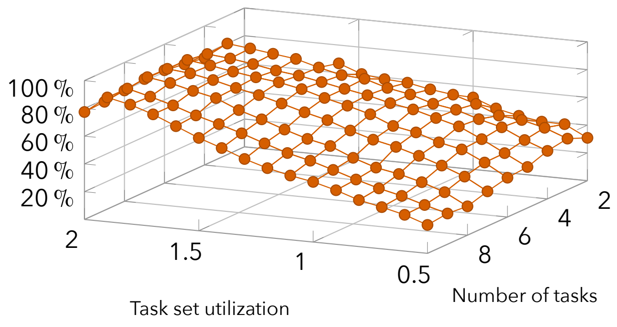
Figure 27: Deadline miss ratio of the Linux Deadline scheduling class on a quad-core system.

In a last experiment, I reduced the maximum per-task utilization to 0.5. I repeated the benchmark for the Deadline scheduling class and Atlas serial queues on a dual-core system. With $m = 2$ and $u_{max} = 0.5$, the theoretical utilization bound for Atlas and the Deadline scheduling class is 1.5.

The result is show in Figure 28.



(a) Atlas serial queue; overload-push load balancing



(b) Linux Deadline scheduling class

Figure 28: Utilization bounds for Atlas serial queues and Linux Deadline scheduling class with a maximum per-task utilization of 0.5 on a dual-core system.

As an example for Atlas I chose the serial queues in the overload-push load balancing mode. Atlas is able to maintain a low number of deadline misses up to the theoretical utilization bound of 1.5, as plotted in Figure 28(a). Atlas serial queues with idle-pull and combined idle-pull/overload-push load balancing show similar deadline miss ratios. I omit the plots of the remaining combinations of Atlas load balancing options for the sake of brevity.

The Deadline scheduling class does not benefit from the reduction in per-task utilization. Figure 28(b) look essentially the same as Figure 24, the plot of the deadline miss ratio of the Deadline scheduling class with per-task utilization of 1.0.

Conclusion & Future Work

I have presented an implementation of the ATLAS paradigm for multi-processor systems. While my implementation is functional, the evaluation revealed several drawbacks of my design. In this chapter I discuss improvements to mitigate the obvious flaws of my ATLAS implementation as well as inspire future research directions concerning the ATLAS infrastructure.

FIRST AND FOREMOST, the utilization bounds of ATLAS have to be improved. While load balancing mitigated the influence of Dhall's effect on serial queues somewhat, a different scheduling algorithm, FPEDF, can avoid Dhall's effect altogether. The challenge here is to find a useful definition of utilization for ATLAS' non-periodic job model. The definition of utilization given in section *The* ATLAS *Task Model* is problematic insofar as that the utilization is time-variant and thus might degenerate into time-slicing, much like the slack in LST scheduling, if two serial queues have similar utilization values. Two possible solutions can be explored, (1) keeping the utilization constant over jobs and (2) the introduction of "utilization inversion", inspired by laxity inversion to mitigate time-slicing in LST.[1]

[1] Oh and Yang, "A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks".

In contrast to serial queues, ATLAS concurrent queues cannot profit directly from a change in scheduling algorithm. The fundamental problem of allowing jobs to be preempted has to be solved first.

A stop-gap solution is to allow multiple worker threads per CPU, where currently exactly one worker thread per CPU is used. The drawback is that a threads needs to be created for each currently active job.

A more sustainable solution is to use user level threads and give each ATLAS job a thread and hence an execution context. If the user level threads are made first-class citizens of the system,[2] preemption and low-cost context switching are possible. First-class status entails that if a user level threads executes a blocking system class, the user level scheduler is informed so that it can execute another user level thread, instead of loosing the execution context because the kernel thread blocked. The kernel needs to inform the user level scheduler of all events which may result in a scheduling decision, such as timer expiration, blocking, and un-blocking of system calls. Communication between kernel scheduler and user level scheduler to improve performance of parallel programs is also proposed by Anderson et al. with their *scheduler activation* mechanism.[3]

[2] Brian D. Marsh et al. *First-Class User-Level Threads*. In: *ACM SIGOPS Operating Systems Review*. Vol. 25. 5. ACM. 1991, pp. 110–121.

[3] Thomas E. Anderson et al. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. In: *Transactions on Computer Systems* 10.1 (1992), pp. 53–79.

Allotting each job its own context allows to use a $(2, 3)$ restricted scheduling algorithm to be used to schedule jobs of ATLAS concurrent queues, so that serial and concurrent queues share a common scheduling algorithm. Furthermore, a per-job execution context is necessary to implement job-splitting, the ATLAS-pendant to task splitting.[4] If systems based on non-periodic job models can indeed profit from job-splitting, as systems based on periodic job models can, is also a question to be answered in future research.

[4] Andersson and Tovar, "Multiprocessor Scheduling with Few Preemptions".

LOAD BALANCING is not as effective, as I have hoped. In overload-push load balancing, continuous sending of IPIs degrades performance even more. A hold-off mechanism to rate-limit IPIs can be used to mitigate that overhead. As an alternative, IPIs can be chained. In chaining, the overloaded CPU sends out only one IPI. If the receiving CPU cannot pull a job, because it is overloaded or close to overload itself, the receiving CPU notifies another CPU, until all CPUs have been tried or the overload situation is resolved. Chaining avoids excessive IPIs and reduces the risk of lock contention on the overloaded run queue.

Idle-pull load balancing offers only a marginal improvement over no load balancing. The reason for this is two-fold. On one hand, a run queue being completely idle is rare. On the other hand, migrating a thread – on which a serial queue is currently based – is rather costly. Instead of idle-pull, "slack-pull", a mode where ATLAS tries to pull jobs when it is in slack time, might prove more beneficial. User space scheduling has the potential to reduce the cost of job migration substantially, and hence make migration more effective.

Placement and migration heuristics should be properly evaluated. In addition to the already-implemented Worst-Fit and Best-Fit heuristics, Next-Fit and First-Fit can be added to ATLAS to measure their performance. Instead of only measuring the deadline-miss-ratio, a more elaborate metric should be chosen. A tardiness of 10 µs is not as bad as a tardiness of 10 ms. But not only the absolute tardiness, or the tardiness relative to the execution time determines the "badness" of a deadline miss. A tardiness of 1 ms for a job with a period of 100 ms could be considered to be not as bad as for a job with a 10 ms period.

USER LEVEL SCHEDULING, in combination with a common scheduling algorithm for ATLAS serial and concurrent queues, has the advantage of a better integration of serial and parallel queues. Currently those constructs merely co-exist instead of complementing each other. With user level scheduling, a pool of worker threads can be used to process jobs from serial and concurrent queues alike, while providing light-weight preemptivity and migration.

WHILE ATLAS IS ABLE TO DETECT OVERLOAD, it currently does not handle overload situations well. First, in overload situations ATLAS should switch from LRT to EDF scheduling. This avoids that jobs miss their deadlines "eagerly", as described in section *Broken Promises*. Furthermore, using signals to inform the user space of deadline misses should be removed. I found the first thing in writing an ATLAS application is installing a signal handler to ignore those signals. Forcing developers to write unnecessary boiler plate code conflicts with ATLAS' ease-of-use principles. Having those signals does not give the user space any information it could not get anyway by way of POSIX' `clock_gettime` with `CLOCK_MONOTONIC`.[5]

[5] Or `std::chrono::steady_clock` in C++11.

Somewhat related to overload management are temporal isolation, fairness and security. Temporal isolation is a property of a scheduler, that no task can cause another task to miss their deadlines. Atlas does currently not have that property. A similar concept is fairness. As long as the machine's capacity is not exceeded, everyone gets enough resources and fairness is not an issue. Only in overload situations, when the machine's capacity is exceeded, some notion of fairness is required. Security is absence of starvation for scheduling layers below Atlas. Atlas threads decay to CFS priority eventually after missing their deadline. This protects the system from starvation if a job enters an infinite loop. If that infinite loop contains a statement to constantly submit new jobs, an Atlas-application might still be able to starve a system. The Deadline scheduling class in Linux handles this problem by using at most 95 % of the system's capacity in its default configuration, leaving 5 % for lower-priority scheduling classes.[6]

[6] Faggioli, Abeni, and Lelli, *Deadline Task Scheduling*.

Instead of updating or cancelling already queued work in overload situations, I propose to queue the minimal necessary amount of work and than add optional work as there is free capacity. The main drawback of reducing the amount of work during runtime is with the prediction component of Atlas. While the execution time was reduced, the workload metrics have not been adapted. If the predictor learns from this tuple of workload metrics and reduced execution time, further predictions will fall short of the actual execution time requirement, exacerbating the overload condition.

The `submit` call could return an estimation of remaining capacity. In the simplest case, this could be a Boolean indicating overload/no overload. Depending on this return value, the application can queue usually small jobs of additional work. A more elaborate scheme is to return an estimation of free CPU time and use the predictor "backwards", translating execution time into a hyper-cube of workload metrics, delimiting combinations of workload metric vectors, whose corresponding jobs would feasibly execute in the remaining CPU time.

A long-term goal is to feature the complete GCD interface, or at least the majority of GCD functions, in Atlas. This would make Atlas a drop-in replacement of GCD, to ease porting GCD-applications to Atlas. On a functional level, this includes incorporation of device I/O. Where GCD is currently only reactive to device I/O, the Atlas scheduler can be extended to be proactive. The Atlas concept can be extended to predict and schedule device I/O, for example for disks and network devices.

A drawback in the current Atlas runtime interface that Roitzsch discovered while porting FFplay to Atlas, is that not every job has a deadline associated with it. The input and decode stages in FFplay are examples for such jobs. The input stage reads data from disk, demultiplexes audio and video and queues jobs for the decode stage. The decode stage decompresses video and audio data and queues the display job. Only the display job has a "natural" deadline – the time when the image must appear on the screen. The input and decode stage do not have a natural deadline; their constraint is to finish early enough for the remaining computation to finish in time. Roitzsch circumnavigated the problem by introducing "artificial" deadlines for the input and decode stage, such that the buffers that link all stages remain approximately half-full. Figure 29 shows the technique of inventing artificial deadlines in pseudo-code.

```
 1 static void read_frame() {
 2   frame *f = new frame;
 3   /* read from disk, demux */
 4   dispatch_async(f->metrics, artificial_deadline, [f]() {
 5       decode_frame(f);
 6     });
 7 }
 8
 9 static void decode_frame(frame *f) {
10   /* decode frame */
11   dispatch_async(f->deadline, [f]() { display_frame(f); });
12 }
13
14 static void display_frame(frame *f) {
15   dispatch_async(artificial_deadline, read_frame);
16
17   while (std::chrono::steady_clock::now() < f->display_time) {
18     std::this_thread::sleep_until(f->display_time);
19   }
20
21   output_to_screen(f);
22   delete f;
23 }
24
25 int main() {
26   dispatch_async(artificial_deadline, read_frame);
27   dispatch_barrier_sync([] {});
28 }
```

Figure 29: Each job requires a deadline; for those jobs that do not have a natural deadline, an artificial one must be invented.

Reading and displaying a frame do not have any metrics because the amount of work is constant for every frame. For reading and decoding a frame, no natural deadline can be given, and artificial deadlines have to invented in lines 5, 15, and 26. It is also noteworthy, that the `frame` structure is not passed as parameter, but is captured by lambdas and implicitly passed from function to function.

Figure 30 shows my proposed solution. `dispatch` calls without deadline return an object of unspecified type on which `then` may be called. The `then` member function accepts lambdas, which in turn accept the return type of the initial function as argument. In case of the example in lines 25 to 32 of Figure 30, this type is `frame *`. The first lambda extracts the workload metrics from its argument, if any.

The second lambda returns the natural deadline, if there exists one. The last lambda contains the work to be performed.

Figure 30: Chaining jobs to avoid inventing artificial deadlines.

```
 1  static frame *read_frame() {
 2    frame *f = new frame;
 3    /* read from disk, demux */
 4    return f;
 5  }
 6
 7  static frame *decode_frame(frame *f) {
 8    /* decode frame */
 9    return f;
10  }
11
12  static void display_frame(frame *f) {
13    while (std::chrono::steady_clock::now() < f->display_time) {
14      std::this_thread::sleep_until(f->display_time);
15    }
16
17    output_to_screen(f);
18    delete f;
19  }
20
21  int main() {
22    while (--frames) {
23      dispatch_async(read_frame)
24          .then([](frame *f) { return f->metrics; },
25                [](frame *f) { decode_frame(f); })
26          .then([] { return nullptr; },              /* no metrics */
27                [](frame *f) { return f->deadline; }, /* deadline */
28                [](frame *f) { display_frame(f); });  /* work */
29    }
30
31    dispatch_barrier_sync([] {});
32  }
```

In case no deadline was given to a call to `dispatch` or `then`, the object of unspecified return type has a private destructor, which yields a compilation error and forces the programmer to eventually complete each such chain with a deadline.

With chaining, execution time prediction is a two-stage process. When the `then` member function in lines 24 and 26 of Figure 30 are executed, metrics cannot be extracted from the frame, because the frame has not yet been read from disc. In the first stage, an estimation based on previous executions of those jobs are made. These preliminary execution time estimations are submitted to the kernel. When the first job, `read_frame`, completes, the first lambda of the next `then` member function is called, to extract the workload metrics. In the second stage, the estimated execution time can be refined using the now-available workload metrics and communicated to the ATLAS scheduler using the `update` system call. In this scheme, the developer is freed from inventing artificial deadlines. Inventing artificial deadlines is just another way of estimating execution times, in this case the execution times of later jobs – a responsibility ATLAS aims to assume from the developer. With chaining, ATLAS estimates the execution time, when no workload metrics are available. When the workload metrics are available, they are used to predict the execution time more accurately.

In contrast to Figure 29, the video player functions in Figure 30 return the `frame` by pointer, so that the result of the function call is directly visible to the caller. Passing the result directly to the caller makes it easier to write unit tests for each function. Implicitly passing data captured by lambdas makes non-intrusive testing of those functions hard to impossible.

ATLAS might further benefit from a vectorized `submit` system call, to submit multiple jobs from a chained dispatch call at once to the kernel. A vectorized `submit` call amortizes the cost of a system call over multiple jobs, allowing ATLAS to schedule even short jobs effectively.

THE SOURCE CODE to both, ATLAS[7] and the ATLAS runtime,[8] is publicly available online. The predictor component, contained in the ATLAS runtime, has been previously published by Roitzsch.[9] To the best of my knowledge, ATLAS is the only implemented system combining scheduling of sets of real-time jobs with a developer-friendly programming interface and overload management.

[7] `https://github.com/hannesweisbach/linux-atlas`

[8] `https://github.com/hannesweisbach/atlas-rt`

[9] `https://github.com/TUD-OS/ATLAS/tree/dissertation`

In this thesis I presented ATLAS-MP*, a multi-processor implementation of the* ATLAS *infrastructure.* ATLAS-MP *consists of serial and concurrent work queues. An integral part of* ATLAS-MP *is load balancing across multiple CPUs for both queue types. I tested my implementation extensively on a single-, dual-, and quad-core system, by loading it with a varying amount of periodic real-time tasks. The presented implementation matches the predicted utilization bound for serial queues, but is still limited by Dhall's effect. Concurrent queues are currently limited by their lack of preemptivity. Regardless of* ATLAS*' current limitations, because of its ability to handle overload situations gracefully,* ATLAS *is able to outperform the Linux Deadline scheduling class in terms of the number of missed deadlines. I proposed solutions to improve the utilization bound of serial and concurrent queues as well as to the* ATLAS *concept in general.*

Bibliography

Abeni, Luca and Giorgio Buttazzo. *Integrating Multimedia Applications in Hard Real-Time Systems*. In: *Proceedings of the 19th Real-Time Systems Symposium*. IEEE. 1998, pp. 4–13.

Anderson, James H. and Anand Srinivasan. *Early-Release Fair Scheduling*. In: *12th Euromicro Conference on Real-Time Systems*. IEEE. 2000, pp. 35–43.

Anderson, Thomas E. et al. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. In: *Transactions on Computer Systems* 10.1 (1992), pp. 53–79.

Andersson, Björn. *Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%*. In: *Principles of Distributed Systems*. Springer, 2008, pp. 73–88.

Andersson, Björn, Sanjoy Baruah, and Jan Jonsson. *Static-priority scheduling on multiprocessors*. In: *Proceedings of the 22nd Real-Time Systems Symposium*. IEEE. 2001, pp. 193–202.

Andersson, Björn and Jan Jonsson. *The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%*. In: *Proceedings of the 15th EuroMicro Conference on Real-Time Systems*. July 2003, pp. 33–40.

Andersson, Björn and Eduardo Tovar. *Multiprocessor Scheduling with Few Preemptions*. In: *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2006, pp. 322–334.

Apple, Inc. *Grand Central Dispatch (GCD) Reference*.

— *Kernel Programming Guide*. 2013. (Visited on 02/02/2016).

Baruah, Sanjoy Kumar. *Optimal Utilization Bounds for the Fixed-priority Scheduling of Periodic Task Systems on Identical Multiprocessors*. In: *IEEE Transactions on Computers* 53.6 (2004), pp. 781–784.

Baruah, Sanjoy K. et al. *Proportionate Progress: A Notion of Fairness in Resource Allocation*. In: *Algorithmica* 15.6 (1996), pp. 600–625.

Baruah, Sanjoy and John Carpenter. *Multiprocessor fixed-priority scheduling with restricted interprocessor migrations*. In: *Proceedings of the 15th Euromicro Conference on Real-Time Systems*. IEEE. 2003, pp. 195–202.

Baruah, Sanjoy et al. *Generalized multiframe tasks*. In: *Real-Time Systems* 17.1 (1999), pp. 5–22.

Bini, Enrico and Giorgio C. Buttazzo. *Measuring the Performance of Schedulability Tests*. In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

Binstock, Atman. *Powering the Rift*. May 2015. (Visited on 12/30/2015).

Birrell, Andrew D. *An Introduction to Programming with Threads*. Tech. rep. 35. DEC Systems Research Center, Jan. 1989.

Blake, Geoffrey et al. *Evolution of Thread-Level Parallelism in Desktop Applications*. In: *SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 302–313.

Brandenburg, Björn B. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis. 2011.

Burchard, Almut et al. *New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems*. In: *IEEE Transactions on Computers* 44.12 (1995), pp. 1429–1442.

Carpenter, John et al. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. In: *Handbook of Scheduling: Algorithms, Models and Performance Analysis*. Ed. by Joseph Y-T. Leung. CRC Press LLC, 2000 N.W. Corporate Blvd., Bocy Raton, Florida 33431.: Chapman & Hall/CRC, 2004. Chap. 30, pp. 30.1–30.30.

Corbet, Jonathan. *CFS group scheduling*. In: *LWN* (July 2007).

Davis, Robert I. and Alan Burns. *A Survey of Hard Real-Time Scheduling for Multiprocessor Systems*. In: *ACM Computing Surveys* 43.4 (2011), p. 35.

Davis, Robert, Attila Zabos, and Alan Burns. *Efficient Exact Schedulability Tests for Fixed Priority Real-time Systems*. In: *IEEE Transactions on Computers* 57.9 (2008), pp. 1261–1276.

Dhall, Sudarshan K. and Chung Laung Liu. *On a Real-Time Scheduling Problem*. In: *Operations Research* 26.1 (1978), pp. 127–140.

Duda, Kenneth J. and David R. Cheriton. *Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler*. In: *SIGOPS Operating Systems Review*. Vol. 33. 5. ACM. 1999, pp. 261–276.

Faggioli, Dario, Luca Abeni, and Juri Lelli. *Deadline Task Scheduling*. Linux kernel documentation.

Faggioli, Dario et al. *An EDF scheduling class for the Linux kernel*. In: *Proceedings of the Real-Time Linux Workshop*. 2009.

Flautner, Kristián et al. *Thread-level Parallelism and Interactive Performance of Desktop Applications*. In: *SIGOPS Operating Systems Review* 34.5 (2000), pp. 129–138.

Funk, Shelby, Joel Goossens, and Sanjoy Baruah. *On-line Scheduling on Uniform Multiprocessors*. In: *Proceedings of the 22nd Real-Time Systems Symposium*. IEEE. 2001, pp. 183–192.

Funk, Shelby et al. *DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling*. In: *Real-Time Systems* 47.5 (2011), pp. 389–429.

Goossens, Joël, Shelby Funk, and Sanjoy Baruah. *Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors*. In: *Real-time systems* 25.2-3 (2003), pp. 187–205.

Hong, Kwang S. and Joseph Y-T. Leung. *On-Line Scheduling of Real-Time Tasks*. In: *Proceedings of the Real-Time Systems Symposium*. IEEE. 1988, pp. 244–250.

Horn, W.A. *Some Simple Scheduling Algorithms*. In: *Naval Research Logistics Quarterly* 21.1 (1974), pp. 177–185.

Imes, Connor et al. *POET: A Portable Approach to Minimizing Energy Under Soft Real-Time Constraints*. In: *Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2015, pp. 75–86.

Kopetz, Hermann. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science & Business Media, 2011.

Korf, Richard E. *A New Algorithm for Optimal Bin Packing*. In: *AAAI-02 Proceedings*. 2002, pp. 731–736.

Lakshmanan, Karthik, Ragunathan Raj Rajkumar, and John P. Lehoczky. *Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors*. In: *21st Euromicro Conference on Real-Time Systems*. IEEE. 2009, pp. 239–248.

Lampson, Butler W. *A Scheduling Philosophy for Multiprocessing Systems*. In: *Communications of the ACM* 11.5 (1968), pp. 347–360.

Lawler, Eugene Leighton. *Optimal Sequencing of a Single Machine Subject to Precedence Constraints*. In: *Management Science* 19.5 (1973), pp. 544–546.

Le Sueur, Etienne and Gernot Heiser. *Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns*. In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. USENIX Association. 2010, pp. 1–8.

— *Slow Down or Sleep, that is the Question*. In: *USENIX Annual Technical Conference*. USENIX Association. 2011.

Liu, Chung Laung. *Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment*. In: *Space Programs Summary*. Vol. II. The Deep Space Network. 37-60. Jet Propulsion Laboratory, 1969. Chap. 3, pp. 28–31.

Liu, Chung Laung and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. In: *Journal of the ACM* 20.1 (1973), pp. 46–61.

Liu, Jane W. S. *Real-Time Systems*. Prentice Hall, 2000.

Lundberg, Lars. *Analyzing Fixed-Priority Global Multiprocessor Scheduling*. In: *Proceedings of the Eighth Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2002, pp. 145–153.

Marsh, Brian D. et al. *First-Class User-Level Threads*. In: *ACM SIGOPS Operating Systems Review*. Vol. 25. 5. ACM. 1991, pp. 110–121.

Mok, Aloysius K. and Deji Chen. *A Multiframe Model for Real-Time Tasks*. In: *IEEE Transactions on Software Engineering* 23.10 (1997), pp. 635–645.

Molnár, Ingo. *CFS Scheduler*. Linux kernel documentation.

Nagle, John. *On Packet Switches with Infinite Storage*. In: *Transactions on Communications* 35.4 (1987), pp. 435–438.

Oh, Sung-Heun and Seung-Min Yang. *A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks*. In: *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*. IEEE. 1998, pp. 31–36.

Oh, Yingfeng and Sang H. Son. *Tight Performance Bounds of Heuristics for a Real-Time Scheduling Problem*. Tech. rep. CS-93-24. 1993.

Phillips, Cynthia A. et al. *Optimal Time-Critical Scheduling Via Resource Augmentation*. In: *Proceedings of the 29th annual ACM Symposium on Theory of Computing*. ACM. 1997, pp. 140–149.

Qualcomm Technologies, Inc. *Snapdragon 810*. (Visited on 10/29/2015).

Roitzsch, Michael. *Practical Real-Time with Look-Ahead Scheduling*. PhD thesis. Technische Universität Dresden, 2013.

Roitzsch, Michael, Stefan Wächtler, and Hermann Härtig. *ATLAS: Look-Ahead Scheduling Using Workload Metrics*. In: *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2013, pp. 1–10.

Srinivasan, Anand and Sanjoy Baruah. *Deadline-based scheduling of periodic task systems on multiprocessors*. In: *Information Processing Letters* 84.2 (2002), pp. 93–98.

Srinivasan, Anand et al. *The Case for Fair Multiprocessor Scheduling*. In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE. 2003, 10–pp.

Standard for Information Technology Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. In: *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)* (Apr. 2013), pp. 1–3906. DOI: 10.1109/IEEESTD.2013.6506091.

Stats, StatCounter Global. *Top 8 Operating Systems from Dec 2014 to Dec 2015*. (Visited on 01/20/2016).

Stoica, Ion and Hussein Abdel-Wahab. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. Tech. rep. TR-95-22. 1995.

Sutter, Herb. *Welcome to the Parallel Jungle*. In: *Dr. Dobb's Journal*. (2012). (Visited on 10/29/2015).

Tanenbaum, Andrew S. *Modern Operating Systems*. 2nd Edition. Upper Saddle River; NJ 07458: Prentice Hall, 2001.

Tia, Too Seng. *Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems*. PhD thesis. University of Illinois at Urbana-Champaign, 1995.

Völp, Marcus, Johannes Steinmetz, and Marcus Hähnel. *Consolidate-to-Idle*. In: *19th Real-Time and Embedded Technology and Applications Symposium*. Vol. 19. Work-in-Progress Proceedings. IEEE. 2013, pp. 9–12.

Wächtler, Stefan. *Look-Ahead Scheduling*. Diploma Thesis. Technische Universität Dresden, 2012.

Zapata, Omar Ulises Pereira and Pedro Mejıa Alvarez. *EDF and RM Multiprocessor Scheduling Algorithms: Survey and Performance Evaluation*. In: *Seccion de Computacion Av. IPN* 2508 (2005).