CABAS: Real-Time for the Masses

Till Smejkal, *TU Dresden*
Jan Bierbaum, *TU Dresden*
Manuel von Oltersdorff-Kalettka, *TU Dresden*
Michael Roitzsch, *Barkhausen Institut*

***Abstract*—Although the real-time community has produced impressive research results, real-time methodology has not gained traction in commodity application development. Only in specialized niches, like avionics and automotive, solid real-time methods are applied because of regulatory requirements and safety concerns. Outside those niches, best-effort-style programming is the norm. We strife to bridge this gap by exposing approachable interfaces to everyday programmers.**

In this paper, we present CABAS, a user-level framework that traces the execution times of jobs online and, based on these data, automatically adapts the parameters of the CBS real-time scheduler inside the Linux kernel. With this approach, we hope to align soft real-time programming with current development methods by freeing the developer from the burden of manually deriving appropriate soft real-time scheduling parameters.

I. INTRODUCTION

The real-time community has accumulated an impressive body of knowledge on task models and schedulability analysis. However, developing a real-time application according to these teachings is a complex undertaking in practice. Therefore, proper real-time methodology is only applied in niches, where regulatory requirements demand a safety certification. From cloud interactive apps to IoT robots and drones, many areas would benefit from solid real-time solutions.

In this work, we present CABAS, a user-level frontend to the constant bandwidth server (CBS) implementation inside the Linux kernel. Offline timing analysis of applications is replaced by automatic online tracing of job runtimes, paired with machine learning to predict execution times. CABAS is inspired by ATLAS [1], which has demonstrated a scheduler design based on tracing of execution times and machine learning to replace worst-case analysis. ATLAS therefore addresses our design goal of simplifying the development of real-time applications, but is limited to single core systems, lacks an approachable programming interface, and requires an in-kernel scheduling component.

After we provide a short overview of CBS and ATLAS, we discuss our framework (Section III). We use synthetic benchmarks and a video player (Section IV) to demonstrate that CABAS is not only able to provide appropriate parameters to the CBS scheduler but also to automatically adapt these parameters when the application's demands change at runtime.

II. BACKGROUND

A. ATLAS *Runtime and Kernel Scheduler*

ATLAS as described by Roitzsch *et al.* [1] consists of two parts. First there are a runtime and programming framework that accept soft real-time jobs from applications and deliver them to the second part, a dedicated soft real-time scheduler. One key aspect of the ATLAS runtime is that programmers do not need to make a complicated execution time analysis of their applications, but only have to specify a relative deadline. The execution time of the application is automatically trained by the runtime using machine learning. This training can be assisted by the programmer by attaching workload metrics to individual jobs which will be considered by the runtime to predict the job's execution time. Thanks to these runtime abstractions, programmers can stay in their respective domains and only have to know which application-specific metrics correlate best with their application's execution time.

However, a significant problem of the ATLAS runtime and scheduler is, that the in-kernel soft real-time scheduler is not part of the mainline Linux kernel. Hence, maintenance of the scheduler is a significant burden, as the Linux scheduling interfaces change regularly. In addition to that, ATLAS cannot simply be used in a plug-and-play fashion, but instead requires a special Linux kernel with the ATLAS soft real-time scheduler built into. Accordingly, although the programming interface and runtime are beneficial for a wide-spread usage of soft real-time, an integration of ATLAS into any system for a normal audience — no kernel developers — is difficult.

B. Real-Time in the Linux Kernel

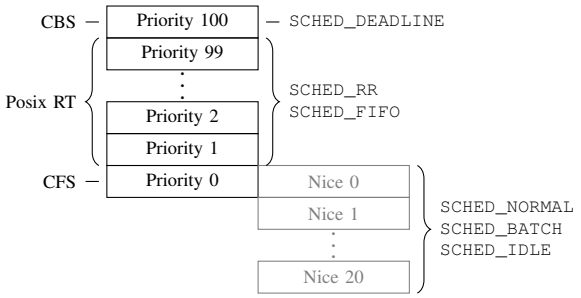


Figure 1: Schedulers of the Linux kernel and their priorities.

The wide variety of use-cases for the Linux kernel and thereby the huge amount of different requirements on the system, resulted in a number of diverse schedulers co-existing within the kernel, as shown in Figure 1. The majority of processes are scheduled using the Completely Fair Scheduler (CFS). CFS handles processes that have no real-time requirements and are thus scheduled with the `SCHED_NORMAL` priority. So-called *nice-levels* can be used to instruct the scheduler to prefer some processes over others, but no guarantees about execution times or when a process is executed are given.

If one strives for more control about a process' execution, the Linux kernel also provides the `SCHED_FIFO` and `SCHED_RR` priorities. Both of them are further separated into 99 sub-priorities and are guaranteed to never be preempted by a process running in a lower priority within `SCHED_FIFO`, `SCHED_RR`, or any process being scheduled by CFS. Although many consider the `SCHED_FIFO` and `SCHED_RR` schedulers already as real-time schedulers, no guarantees among the processes are given by the scheduler regarding, for example, the process' time to finish. Instead, processes are just executed in the order in which they were presented to the scheduler. However, the Linux kernel also supports yet another scheduling priority, namely `SCHED_DEADLINE`. Processes in this priority are handled by a Constant-Bandwidth-Server-like scheduler and are always executed with a priority above any other scheduler in the system.

Constant Bandwidth Server (CBS) [2] is a scheduling algorithm that handles hard and soft real-time tasks simultaneously, guaranteeing deadlines for hard real-time jobs and a constant bandwidth for the soft real-time ones. The CBS implementation in Linux supports hard and soft real-time tasks. In both cases, a task τ_i is defined by its inter-arrival time T_i and its execution time Q_i. For hard real-time tasks, T_i and Q_i refer to the minimum inter-arrival time and the worst case execution time (WCET), whereas for soft real-time tasks the parameters are just considered mean values. CBS will schedule its workload using Earliest Deadline First (EDF). For hard real-time tasks the inter-arrival time T_i is used as the deadline of individual jobs. For soft real-time tasks the Constant Bandwidth Server protocol is used to manage the jobs and calculate their per-job deadlines. Every Constant Bandwidth Server maintains a budget q_i that is expended when a corresponding job executes. Once the budget reaches 0, the deadline of the server is increased by T_i of the soft real-time task that the server manages.

In order to integrate CBS in the Linux kernel, various changes to its behavior were proposed and implemented [3, 4]. Whereas in the original CBS description the budget of a server depleted equally to the executed time of the corresponding job, the implementation in the Linux kernel uses a varying rate based on the state of the server and the overall utilization of the system. Furthermore, Linux distinguishes between active servers — servers that have pending jobs — and inactive ones — servers without pending or running jobs. This state of a server influences the overall utilization of the system and hence the rate at which servers decrease their budget. In addition, the Linux implementation of CBS always leaves some space in its real-time schedule to let other processes that are scheduled with lower priorities to also execute on the CPU.

C. Soft vs. Hard Real-Time

The design of the ATLAS scheduler and framework is tailored towards soft real-time applications. CBS on the other hand supports both hard and soft real-time applications. The nature of ATLAS with its predictor, that might estimate incorrect execution times is an inherent problem for hard real-time applications, which are usually used in mission-critical systems where missing a deadline would lead to a potentially catastrophic failure. For soft real-time applications a deadline miss is usually not critical but just not favorable. Typically, with soft real-time applications the usability of a result decreases the later the job finishes and hence might result in unwanted glitches or visible artifacts. Yet, normally soft real-time applications can still continue even after a deadline miss. Hence, our work CABAS focuses on making the creation of soft real-time applications easier since moving the time-consuming execution time analysis of an application into an online-trained predictor is acceptable. Deadline misses, which might occur especially at the beginning of the predictor training, can be tolerated by the application. However, for hard real-time applications CABAS is probably not suited, although the used Linux CBS scheduler is capable of managing such scenarios as well.

III. IMPLEMENTATION

A. Using Linux CBS

As described in Section II-B the Linux kernel scheduler comprises multiple different schedulers that work together, each handling a different scheduling priority. In order to run real soft or hard real-time applications on a Linux system one has to instruct the kernel to use the `SCHED_DEADLINE` scheduler for this particular application via the `sched_setattr` system call. To define and possibly change the task parameters such as its WCET or its deadline one has to use the additional `sched_setparam` system call.

As the Linux schedulers always schedule individual threads (`struct task_struct` within the kernel), it is even possible to compose an application of normal threads managed by CFS and real-time threads managed by the Linux CBS scheduler. However, this thread-based management within the kernel makes the integration of the task- and job-based model of real-time applications difficult. Especially in a soft real-time scenario, where there are multiple possibly independent jobs, the mapping to separate threads can be burdensome. Our framework CABAS addresses this problem and provides the programmer with an easy-to-use interface that abstracts thread creation, configuration, and management away.

B. The CABAS Framework

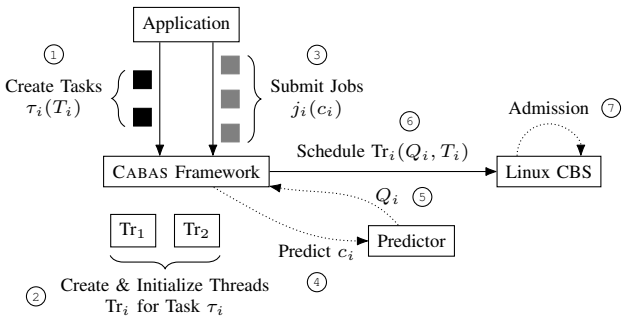


Figure 2: The architecture of the CABAS framework.

The architecture of CABAS is visualized in Figure 2. In general CABAS consists of the framework, which handles the interaction with the kernel's CBS scheduler as well as manages all the necessary state, and the estimator, which is used to predict missing information such as a job's execution time based on workload metrics provided by the application. In order to run a real-time workload using CABAS, the programmer first has to create tasks τ with an expected inter-arrival time T_i ①. For each task CABAS will create a corresponding thread and initialize the thread such that it can be run with the Linux CBS scheduler ②. This initialization includes, for example, the calls to `sched_setattr` and `sched_setparam` with the correct parameters. Later, when the application submits actual jobs to the framework, it can optionally augment them with metrics c_i that correlate positively with the jobs' runtime ③. CABAS uses the workload metrics — if provided — to predict the execution time Q_i of the job based on observations of previous job executions ④ & ⑤. We use the linear least-squares auto-regressive predictor presented by Roitzsch *et al.* [1], which we found to work reasonably well for a set of synthetic benchmarks and a video player; further discussed in Section IV. As shown in the figure, the predictor is not tightly integrated into the CABAS framework and can thus be substituted by the programmer with a specialized implementation. The prediction step can also be omitted when the programmer provides the expected execution time with the job submission. Providing an upper bound from a worst-case execution time analysis allows hard real-time operation, but this is not the focus of our work.

In any case, CABAS will then schedule the thread corresponding to the job's task with the specified parameters on the kernel's CBS scheduler ⑥. In the case that the predictor determined different parameters then for the previous run, CABAS will update the scheduling parameters via the `sched_setparam` system call and the kernel's CBS scheduler will rerun its admission test ⑦ in order to prevent over-utilization of the system. When this admission should fail, the framework will receive an error upon which further steps need to be taken by the programmer. Ignoring the error and running the task with the old parameters might be a valid option, but overload handling can be application-specific and is orthogonal to our solution. If multiple jobs are submitted for one task, the framework will internally queue them and start them either at a specified point in time or as soon as possible.

C. Programming with CABAS

```
1 int create_task(int period, void (*execute)(void *), int
      exec_time);
2 int create_task_pred(int period, void (*execute)(void *),
      struct metrics (*generate)(void *));
3 void add_job_to_task(int task, void *arg);
4 void join_task(int task);
```

Listing 1: The C Interface of the CABAS Framework

CABAS itself is an open-source C/C++ framework[1] and can thus be used by many existing applications. Even integrating it into modern programming languages like Rust is easily doable as foreign function calls to C libraries are usually supported. Listing 1 shows the C-interface of CABAS, which consist of mainly three important functions. There are `create_task_pred` and `create_task` to submit new tasks to the framework. Whereas the former version will instruct the framework to use the estimator, the latter follows the more traditional task model for CBS where a median execution time and inter-arrival time for a task's jobs are specified at creation time. When the predictor is used, CABAS will call the `generate` function before every job execution. That function should return the workload metrics for the given job, which are then used by the framework to predict the job's execution time. Since this function can be specified by the programmer, various techniques are possible for generating the metrics. To run the actual job, the framework will call the `execute` function specified at the task creation and provide it the corresponding arguments for the job. CABAS assumes that a task can be represented by one function. Each job incarnation is a call to this function with varying arguments. Accordingly, the third function that the framework provides (`add_job_to_task`) adds new jobs to a task by specifying the arguments with which the `execute` function should be called. The interface function `join_task` will wait for the completion of all jobs of one task.

D. CABAS *vs. Traditional CBS*

One significant difference between CABAS and traditional CBS is that CABAS dynamically adjusts the scheduling parameters during the execution of the application. In a traditional CBS soft real-time application, a programmer would use offline analysis to determine the mean execution time and the mean inter-arrival time of a task and provide these values to the scheduler. Whereas CBS itself is designed to handle variations within the execution times of the jobs, significant changes in the behavior of jobs, as they can happen with phased applications, can lead to a significant tardiness (see Section IV for an example). CABAS however can alleviate such effects by dynamically adjusting the CBS scheduling parameters at runtime and is thereby able to react to changing application behavior. For example, if an application enters a phase where jobs take longer than usual, CABAS will automatically increase the budget for the corresponding CBS Server, reserving more CPU time for the job.

This dynamic parameter adaptation is also beneficial when an application is ported to new hardware. With traditional CBS one either has to do a new analysis to calculate the mean execution times of the application on the new hardware, or use the previously calculated values and accept a potentially significant job tardiness. With CABAS and its predictor no analysis is necessary as the system will automatically learn the mean execution times and use them accordingly.

[1] https://github.com/TUD-OS/Cabas

IV. Evaluation

We evaluate CABAS with two different applications: a synthetic benchmark and an elementary video player. All experiments were performed on a machine with an Intel Core i7-4790 (4 cores, 2 threads each) and 8 GiB RAM running Ubuntu 21.10 with Linux 5.13. For collecting runtime data, we used the low-overhead *Linux Trace Toolkit: next generation* (LTTng) to set a tracepoint to the end of a job and calculated its tardiness. The time CABAS requires for training and estimation is accounted as part of a job's execution time.

A. Synthetic Benchmark

A synthetic soft real-time workload comprises multiple tasks, each with a target mean inter-arrival time and mean execution time. The values for individual jobs use a standard distribution with the chosen mean value and a standard deviation of 10%. Within this range the values are uniformly distributed.

To acquire representative and reliable results, we generated 1000 workloads for each integer utilisation in the range between 50% and 200% using the *UUniFast* algorithm [5]. Each workload comprises five tasks with at least five jobs each, capping the inter-arrival time at 10 000 µs. The minimum inter-arrival time is set to 200 µs. Workloads run for at least 50 000 µs on either one or two cores. Utilisation above 100% were only run in the two-core scenario as CBS is not expected to handle overload situations. A dedicated thread in `SCHED_FIFO` submits the real-time jobs to the CABAS framework, which then executes them on Linux CBS. The jobs themselves consist of a busy loop consuming the job's appointed runtime.

Figure 3 shows the mean tardiness of a job across the 1000 workloads for a specific system utilisation. CBS is statically configured with the target mean inter-arrival time and mean execution time of a workload. Note, that in a real-world scenario the application developer would need to determine and provide this information. CABAS, on the other hand, is given only the mean inter-arrival time and automatically derives the expected runtime using its predictor (see Section III).

The tardiness of jobs increases up to 100 µs to 250 µs when scheduled with CABAS. The overall behaviour is similar for the dual core scenario. Considering the simplistic nature of the synthetic real-time jobs, the only possible metric is the target runtime itself. Knowing these values, however, would allow to directly use CBS and thus defeat the purpose of CABAS, so we did not use any metrics.

B. Video Player

A typical application that can profit from soft real-time is a media player that wants to provide smooth playback even in the presence of high system load. Unfortunately, adopting real-time scheduling is complex and no major media player makes use of it. To evaluate this interesting scenario anyway, we developed an elementary video player based on *FFmpeg*. The player reads a media file, extracts the video stream, decodes the individual video frames, and renders them using *SDL2*. Any other media streams (audio, subtitles, …) are discarded.

The player maintains three processing tasks: (read and) *decode* frames, *scale* the frame to the expected output size and colour format, and *render* the frame at the right point in time. In contrast to regular video players, ours never drops late frames to highlight the effect of such delays. For the evaluation, we used the animated short film "Big Buck Bunny"[2]. This video is encoded with H.264, has a resolution of 1920×1080 pixels, and a frame rate of 60 Hz. We played the first 3 minutes of the video, i.e. 10 800 frames. Given the target frame rate, all tasks have a period of $\frac{1}{60}$ s. A dedicated management thread, pinned to one core, issues new jobs to the aforementioned tasks and forwards data whenever appropriate. Processing jobs are free to use any of the remaining hardware threads.

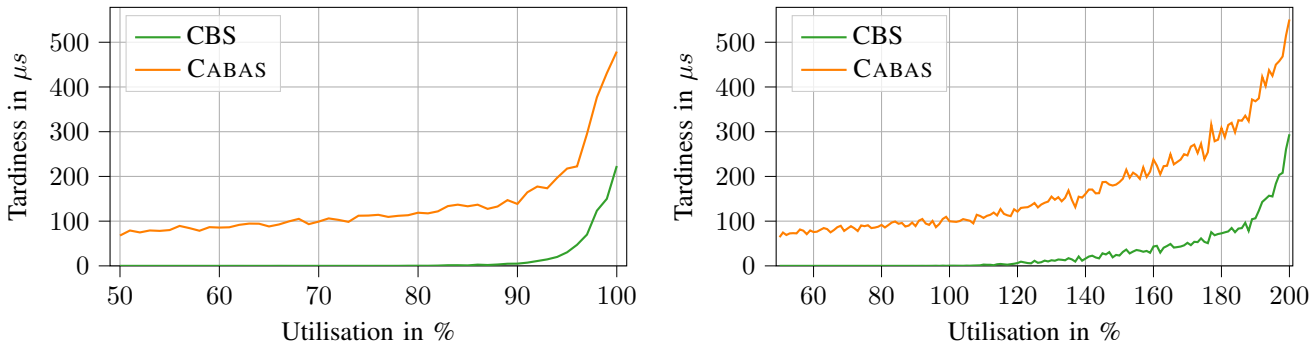
CABAS allows to provide application-specific metrics with a new job, which are then used to improve the runtime prediction (see Section III). Whereas scaling and rendering have a homogeneous workload and take a fixed amount of time, this is different for the decoding task. Due to the nature of modern video codecs, frames might have to be decoded out of order: Decoding a *B frame* (bidirectional coded picture) requires that its successor has been decoded. Thus, the execution time for decoding the chronologically next frame can vary depending on that frame's type and we use this frame type as a metric for CABAS. To give the scheduler more leeway in compensating for deviations, we established a buffering scheme that allows up to 8 scale jobs and 8 render jobs to exists in the system. In contrast to the synthetic benchmark, the mean job runtimes for the video player tasks were not known. We determined these times in an initial run of the player in CFS on an unloaded system. The decoding time for each frame is depicted in Figure 4. To simplify the player's implementation, we also pre-determined the frame type during this initial run.

Users of a media player care about smooth playback, i.e. the timely rendering of frames. Figure 5 shows mean tardiness for each individual frame when using different schedulers. On a lightly loaded system CFS shows no tardiness at all, which also demonstrates that the CPU is capable to decode all frames fast enough to maintain the target frame rate. However, with heavy load, CFS finishes with the video playback almost 9 seconds late. A heavily loaded system for CFS was simulated by running in parallel to the video player as many `stress`[3] benchmarks as the system has CPUs.

CBS, configured with the mean value of all execution times for each specific task, accumulates a delay of 18 seconds over the 3 minute runtime of the video. Further tests showed that long-running decode jobs may leave the scale task without work and thereby depleting the scale task queue eventually. Since the scale task has a very stable runtime, its budget is just enough to complete one job per period. Therefore, the scale task can never catch up with the refilled queue once it was initially delayed. Unfortunately, this creates a downstream effect as the following render task will also eventually deplete its queue and be limited by the throughput of the scale task.

[2] https://peach.blender.org/download/

[3] https://github.com/resurrecting-open-source-projects/stress



(a) Real-time tasks on a single core

(b) Real-time tasks on two cores

Figure 3: Mean tardiness of a job depending on the workload's utilisation.

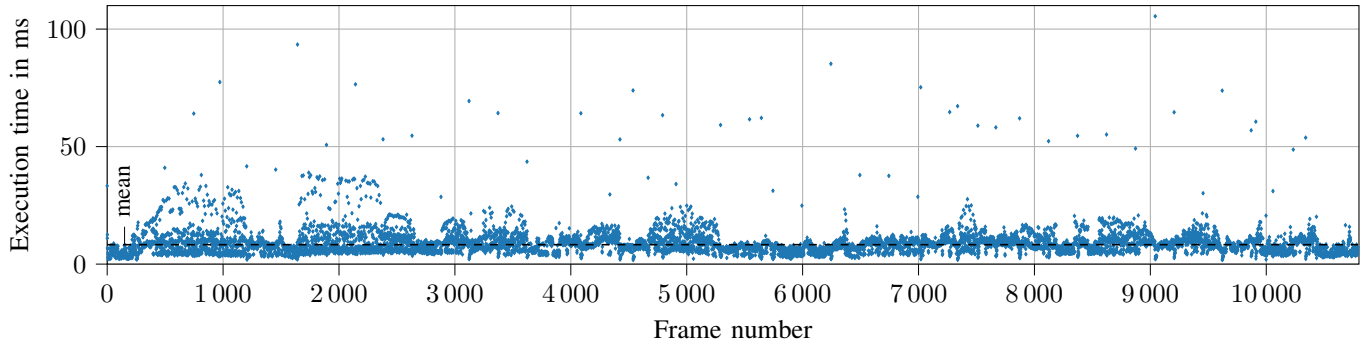


Figure 4: Execution times of the *decode* task in low-load CFS. The black line shows the mean value used for CBS (8.268 ms).

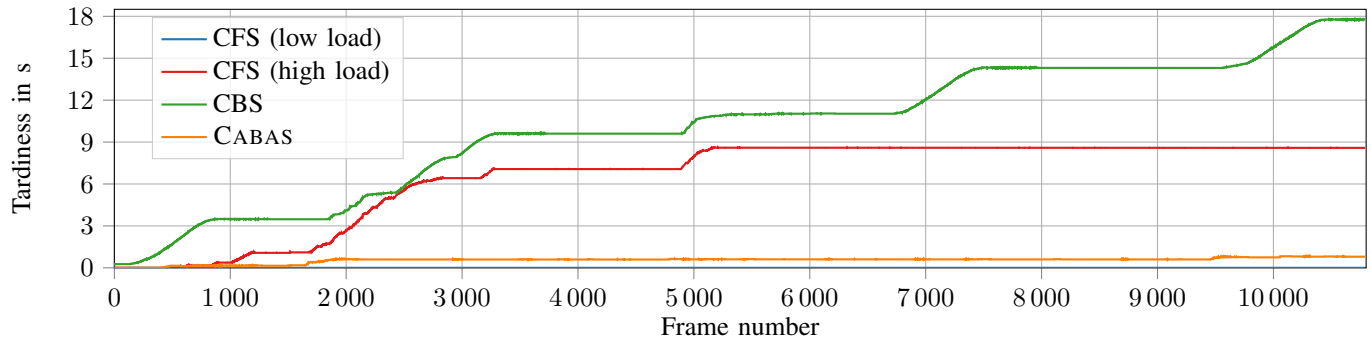


Figure 5: Tardiness of the *render* task for individual frames.

The main problem is that the Linux CBS implementation is not work conserving in contrast to when the player runs in CFS. Further measurements showed that overbudgeting the scale task (e.g. giving it twice the budget) can remedy this situation to some degree, but would not add any additional insight to the comparison of the schedulers. Such a decision is, once again, on the programmer of the application. In addition we found out, that the rendering of the video in the SDL2 player window done by the X-server also introduces some push back on the render task. The main problem is that the X-server itself is not scheduled using real-time priorities. This mismatch of scheduling priorities between the render task and the X-server creates unpredictable additional delay in the render task as the task waits for the acknowledgment of the window manager that the window was updated before it renders another frame. We could verify with additional measurements that not drawing the frame on the X-server window but just in an internal frame buffer reduces the delay of the video playback when run with CBS. When this drawing just to an internal frame buffer is combined with the aforementioned overbudgeting of the scale task, the video player is even able to catch up with the playback eventually after a delay happened and will thus finish with a lower or even no delay in contrast to what is shown in Figure 5 for CBS

When running the player with CABAS tardiness increases, too, but less drastically than with CBS. Adjusting the scheduling parameters by learning and estimating the execution times with CABAS' predictor can better compensate for the long-term variation in execution times. CABAS will temporarily give, for example, the scale tasks more budget when a long-running decode occurs in the decode task, which causes a delay of the scale task. The scale task is thus able to catch up with the refilling queue eventually. When the scale task catches up with the new load and has a stable execution time again, CABAS will reduce the budget accordingly. To sum it up, due to the execution time predictor, CABAS can react to changes in the application behavior which otherwise have to be done manually by the programmer. Additional measurements with CABAS where we disabled the rendering of the frame in the X-server window, similar to the scenario for CBS, resulted in a tardiness of 0 s and show that our framework is able to smoothly play back a video even without any additional scheduling-specific knowledge (e.g. fine-tuning of scheduling parameters) from the programmer.

V. Related Work

CABAS intends to make real-time scheduling more accessible for developers. A considerable expertise barrier exists when application developers implement problems amenable to real-time scheduling, which has also been observed by Brandenburg [6]. Other works share our goal of lowering this barrier, approaching it either using reservation-based methods or the fair-share schedulers present in commodity systems.

a) Reservation Approaches: The original CBS [7] wraps tasks with varying execution times in a server to make scheduling behavior more robust if task parameters are not exact. To improve quality of service for a soft real-time load, later work added dynamic changes to the allocated server bandwidth [8] and slack reclaiming mechanisms [9]. CBS-based schedulers have also been demonstrated on multicore systems [10] and within frameworks for quality of service control [11].

At their core, all reservation-based mechanisms require the developer to report an execution time or bandwidth requirement to the scheduler. Such information is difficult to obtain for highly workload-dependent tasks that end-users run on a variety of hardware platforms with different speeds. The added adaptivity features, however, enable CBS to tolerate over- as well as underspecification. In return, CBS provides timeliness guarantees and is suitable for hard real-time. CABAS relieves the developer from determining execution times entirely, but is not suitable for hard real-time. A common trait of CBS'es adaptive reservations and CABAS is the use of a per-task execution time estimator. However, the estimator presented for CBS [8] extrapolates by using only past execution times. CABAS leverages workload metrics to improve estimations and transparently communicates execution time information to the CBS scheduler without involving the developer.

b) Fair Processor Sharing: Fair-share schedulers allow limited control over scheduling behavior by providing a priority interface like the Unix nice levels. However, a developer cannot determine the correct priority level for an application without a complete overview of all other applications in the system. Borrowed Virtual Time [12] expresses task priorities with a concept called *warp time*, but assigning these parameters still requires global system knowledge, because warp times are not intrinsic to one application. CABAS, CBS, and all other EDF-based schedulers employ deadlines which are parameters from the problem domain of the application. They can be specified without global knowledge.

VI. Conclusion

We present CABAS, an easy-to-use user-level frontend to the CBS real-time scheduler. CABAS mediates between application and scheduler, enabling developers to spawn new work items with individual execution behavior by way of a single function call. Developers need to reflect only on application-local behavior and CABAS only asks for parameters from the application domain: Periodic deadlines express latency requirements, workload metrics describe jobs. CABAS uses these values and execution time information it gathers in the background to derive scheduling parameters for CBS.

We have demonstrated, using synthetic benchmarks and video playback, that CABAS can properly handle common use cases. Not only can it derive suitable CBS parameters, but also automatically adjust them to changing application demands at runtime. Our framework, thus, makes soft real-time scheduling readily available to the average developer.

As a next step we plan to look more closely into why CBS has such a devastating behavior for the video player example and whether we can improve CABAS and Linux' CBS in general to closer match the CFS low load performance. Furthermore, we want to investigate other applications and areas where CABAS can be applied. One interesting aspect would be to use the real-time-specific information such as deadline and estimated execution time for more energy-aware scheduling.

References

[1] M. Roitzsch *et al.*, "ATLAS: Look-ahead scheduling using workload metrics," in *Proceedings of the 19th IEEE real-time and embedded technology and applications symposium*, ser. RTAS, Philadelphia, PA, USA: IEEE, Apr. 2013, pp. 1–10, ISBN: 978-1-4799-0184-5. DOI: http://dx.doi.org/10.1109/RTAS.2013.6531074. [Online]. Available: http://os.inf.tu-dresden.de/papers_ps/rtas2013-mroi-atlas.pdf.

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE real-time systems symposium (cat. no. 98CB36279)*, IEEE, 1998, pp. 4–13.

[3] L. Abeni *et al.*, "Greedy CPU reclaiming for SCHED_DEADLINE," in *Proceedings of the real-time Linux workshop (RTLWS), dusseldorf, germany*, 2014.

[4] J. Lelli *et al.*, "Deadline scheduling in the Linux kernel," *Software: practice and experience*, vol. 46, no. 6, pp. 821–839, 2016.

[5] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, May 2005. DOI: 10.1007/s11241-005-0507-9.

[6] B. B. Brandenburg, "The case for an opinionated, theory-oriented real-time operating system," in *1st international workshop on next-generation operating systems for cyber-physical systems*, ser. NGOSCPS, Montreal, Canada, Apr. 2019. [Online]. Available: https://www.cse.wustl.edu/~cdgill/ngoscps2019/papers/NGOSCPS2019_Brandenburg.pdf.

[7] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE real-time systems symposium*, ser. RTSS, Madrid, Spain: IEEE, Dec. 1998, pp. 4–13, ISBN: 0-8186-9212-X. DOI: http://dx.doi.org/10.1109/REAL.1998.739726. [Online]. Available: http://retis.sssup.it/~giorgio/paps/1998/rtss98-cbs.pdf.

[8] L. Abeni *et al.*, "QoS management through adaptive reservations," *Real-time systems*, vol. 29, no. 2, pp. 131–155, Mar. 2005, ISSN: 0922-6443. DOI: http://dx.doi.org/10.1007/s11241-005-6882-0. [Online]. Available: http://retis.sssup.it/~lipari/papers/real_time_systems_cucinotta_palopoli_adaptive_reservations.pdf.

[9] L. Palopoli *et al.*, "Weighted feedback reclaiming for multimedia applications," in *Proceedings of the 2008 IEEE/ACM/IFIP workshop on embedded systems for real-time multimedia*, ser. ESTImedia, Atlanta, GA, USA: IEEE, Oct. 2008, pp. 121–126, ISBN: 978-1-4244-2612-6. DOI: http://dx.doi.org/10.1109/ESTMED.2008.4697009. [Online]. Available: http://disi.unitn.it/~palopoli/publications/estimedia08.pdf.

[10] S. Kato *et al.*, "AIRS: Supporting interactive real-time applications on multicore platforms," in *Proceedings of the 22nd euromicro conference on real-time systems*, ser. ECRTS, Brussels, Belgium: IEEE, Jul. 2010, pp. 47–56, ISBN: 978-0-7695-4111-2. DOI: http://dx.doi.org/10.1109/ECRTS.2010.33. [Online]. Available: http://ertl.jp/~shinpei/papers/ecrts10.pdf.

[11] T. Cucinotta *et al.*, "On the integration of application level and resource level QoS control for real-time applications," *IEEE transactions on industrial informatics*, vol. 6, no. 4, pp. 479–491, Nov. 2010, ISSN: 1551-3203. DOI: http://dx.doi.org/10.1109/TII.2010.2072962. [Online]. Available: https://scholar.google.com/scholar?cluster=5484267806271076788.

[12] K. J. Duda and D. R. Cheriton, "Borrowed-Virtual-Time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proceedings of the 17th ACM symposium on operating systems principles*, ser. SOSP, Charleston, SC, USA: ACM, Dec. 1999, pp. 261–276, ISBN: 1-58113-140-2. DOI: http://doi.acm.org/10.1145/319151.319169. [Online]. Available: http://gregorio.stanford.edu/bvt/bvt.ps.