ATLAS: Look-Ahead Scheduling Using Workload Metrics

Michael Roitzsch Stefan Wächtler Hermann Härtig
Operating Systems Group
Technische Universität Dresden
Dresden, Germany
Email: {mroi,waechtler,haertig}@os.inf.tu-dresden.de

***Abstract*—From video and music to user interface animations, a lot of real-time workloads run on today's desktops and mobile devices, yet commodity operating systems offer scheduling interfaces like nice-levels, priorities or shares that do not adequately convey timing requirements. Real-time research offers many solutions with strong timeliness guarantees, but they often require a periodic task model and ask the developer for information that is hard to obtain like execution times or reservation budgets. Within this design space of easy programming, but weak guarantees on one hand and strong guarantees, but harder development on the other, we propose ATLAS, the Auto-Training Look-Ahead Scheduler. With a simple yet powerful interface it relies exclusively on data from the application domain: It uses deadlines to express timing requirements and workload metrics to express resource requirements. It replaces implicit knowledge of future job releases as provided by periodic tasks with explicit job submission to enable look-ahead scheduling. Using video playback as a dynamic high-throughput load, we show that the proposed workload metrics are sufficient for ATLAS to know an application's execution time behavior ahead of time. ATLAS' predictions have a typical relative error below 10%.**

***Keywords*-auto-training look-ahead scheduler; self-describing jobs; workload metrics**

I. INTRODUCTION

Today's desktops and mobile devices host many real-time workloads, ranging from continuous media like audio and video playback to user interface animations which are used in applications to improve usability. Most of these real-time workloads are implemented as best-effort tasks, leading to the undesirable situation that the CPU scheduler is largely unaware of the real-time nature of the applications it manages. The default Linux scheduler CFS (Completely Fair Scheduler) for example provides equal CPU shares to tasks on the same priority. In doing so, it may violate timing constraints of applications simply because the application has no interface to expose them.

In Figure 1, we illustrate the quality of video playback by plotting the interval between adjacent frames. After one minute of playback, a competing load of ten CPU-hogging processes starts. Unaware of the video player's timing requirements, CFS will now assign it less than 10% of total CPU time. Consequently, the video becomes unpleasant to watch. This experiment was conducted playing the Hunger Games trailer (see Table I) on a 2.4 GHz Intel Core i5, running Ubuntu Linux 12.04.1 LTS. CFS auto-grouping was enabled, but ineffective, because the competing load comprised ten independent processes next to the video player.

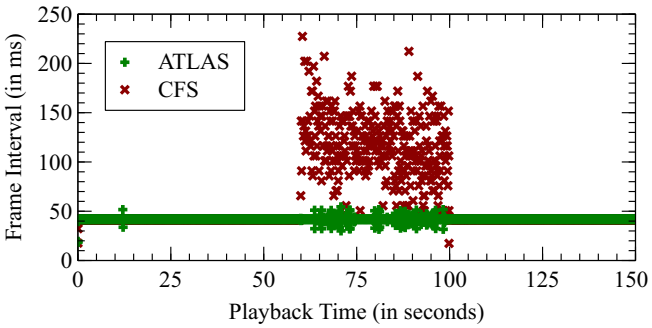

Figure 1. Video Smoothness with Competing Load under CFS and ATLAS

CPU schedulers commonly offer a priority-based mechanism, for example the traditional Unix nice levels. However, it is difficult for developers to express the timing requirements of an application using such an interface, because to decide on the appropriate priority level, the developer needs a global system view.

The upside is that developers do not need to worry about real-time programming. They can freely structure their applications without being forced into a specific task model. Real-time research offers a different approach: In its most rigid incarnation, it imposes a periodic task model on applications and asks developers to provide the maximum execution time needed within each period. Such information is hard to obtain, because it depends on the end-user hardware and the executed workload. The corresponding scheduler however can offer strong and mathematically proven timeliness guarantees.

Between these two ends lies a spectrum of interesting scheduler interfaces from both systems and real-time research, which we discuss in Section VI. In this paper, we explore one point in this design space which we dub ATLAS, the *auto-training look-ahead scheduler.* We investigate what timing information we can obtain while trying to curb developer effort. ATLAS relies only on deadlines to express timing

requirements and on *workload metrics* to express resource requirements. It uses *explicit submission* of future jobs instead of periodic tasks. Figure 1 shows how ATLAS behaves in the previous experiment. Video stutters are noticeably reduced, but the developer needs to adapt the player code to use the ATLAS interface.

Contribution

We propose ATLAS, a task model and scheduler interface for soft deadlines. We do not contribute improvements in scheduling theory, but in the interaction between application and scheduler. Other than the majority of related work, ATLAS does not employ a periodic task model. Explicit submission of future jobs substitutes the implicit knowledge on future execution that periods provide. The scheduler interface as seen by the application developer only asks for parameters within the application domain. Deadlines specify timing requirements; resource requirements are specified using workload metrics: parameters from the workload that describe its computational weight. Machine-learning is used to automatically derive execution time estimates from the metrics *before the actual execution*. This prediction method is based on previous work [1], but here we remove the need for a dedicated training phase. We will detail the architecture and interfaces in Section III and explain the workload metrics in Section IV.

We claim that the flexibility of this task model allows to inform the scheduler more accurately of application behavior than alternative approaches. We validate this claim in Section V by demonstrating that the scheduler can predict the future execution of applications. Outside the implicit clairvoyance of periodic task systems, no other scheduling system we are aware of features a comparable look-ahead capability.

II. DESIGNING ATLAS

The target environment of ATLAS is the interactive desktop. It is not suited for reactive systems that continuously supervise sensors and actuators, for example in an industrial control environment or within other deeply embedded systems. We believe an interactive system should not reject the user's instruction to start an application. Therefore, a task admission is not appropriate and consequently, the system does not handle hard deadlines. Overload situations may occur and ATLAS needs to handle them. Because it predicts future application behavior, ATLAS can detect and react to overload before it occurs. However, overload mitigation strategies are not the focus of this paper. Section VII provides a summary of our ideas. Similarly, we only discuss ATLAS running on a single processor, with a sketched multicore extension following again in Section VII.

A. Example Application

We demonstrate our ideas with video playback as an example workload. We think video is a good example, because it presents intrinsic timing requirements, it requires heavy computation and the load is highly dynamic. Thus, it constitutes a super-set of typical demands of desktop real-time applications. We do not see ATLAS as a video-only scheduler, but we use video to motivate some design decisions. Therefore, we now take a closer look on video player behavior and architecture.

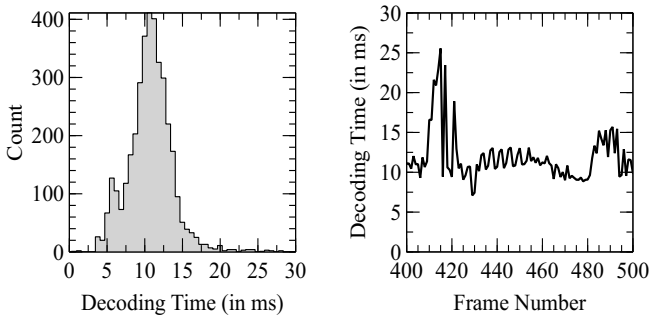

Figure 2. Frame Decoding Times for Test Video "Hunger Games" (see Table I).

With modern video decoding standards like H.264 [2], the decoding times for individual frames highly depend on the input bitstream. Figure 2 illustrates this variability aggregated over an entire clip and shows the short-term fluctuations even between adjacent frames. This behavior is a challenge for scheduling, because many task models assume constant execution time for recurring jobs. The long-tail distribution of frame decoding times however renders worst-case planning impractical as it would result in over-provisioning.

B. Player Architecture

We use the FFmpeg decoder library [3] in version 0.11.1 and its included video player FFplay for experiments. During playback, FFplay employs three threads: The first reads the video from disk and splits it into individual audio, video and subtitle packets. The second thread is responsible for decoding the compressed video data into video frames, while the third thread triggers the refresh of the video picture on screen. Audio and subtitle decoding would be performed in two additional threads, but we ignore those cases here because they do not provide additional insight.

All player stages must cooperate to display the final frames in a timely fashion. Therefore, we will have all three threads scheduled by ATLAS. Within one application, we therefore handle an I/O-bound thread (video input), a CPU-heavy thread (video decoding) and a time-critical control thread (display refresh) simultaneously. The three stages are connected by FIFO queues, which help to even out the dynamic behavior of video decoding, but they also constitute a dependency between the stages. A thread will block if it does not find sufficient data on its incoming queue. A free slot on the outgoing queue is necessary to store the produced result.

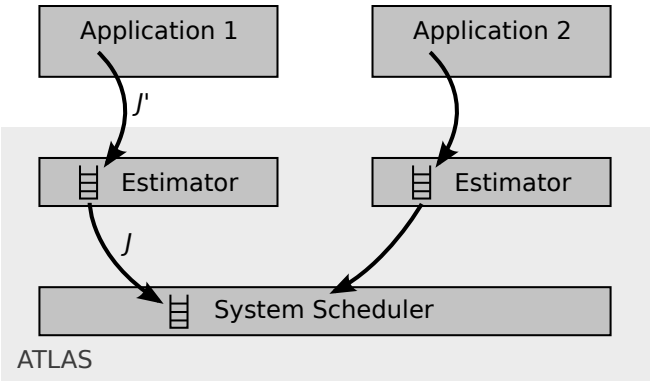

Figure 3. ATLAS Architecture

C. Look-Ahead

The queueing also provides the opportunity of a limited look into the future. With data yet to be decoded already present in the player's buffers, the application already knows what its next jobs will be. This knowledge should be made available to the scheduler to allow it to foresee the future behavior of the application and construct its schedule accordingly. Global aggregation of such look-ahead knowledge enables the scheduler to anticipate and react to deadline misses before they occur.

We think this limited look into the future is a common trait of many desktop real-time workloads: The user triggers an application once — by clicking the play button for example — and the application will then autonomously process a predetermined real-time task. The next jobs do not depend on external events like in reactive systems, but the upcoming work is completely laid out in the application's code and determined by its input data. All media players, video editing software, and desktop animations behave this way. This even extends to video conferencing: buffers are shorter to reduce latencies, but the general construction is the same.

The counter-examples are applications that react with very low latencies to repeated user input. This includes action games and software instruments in music applications. Here, the user input directly triggers each real-time job individually, so no buffering in the application can build up to provide knowledge of future jobs. ATLAS can still be used to schedule those applications, but the look-ahead horizon is reduced.

III. SCHEDULER INTERFACES

In this section we describe the interface between applications and ATLAS. As established in the previous section, ATLAS allows applications to expose knowledge about future job executions. In order for the interface to be easy to use, we only want to ask developers for data from the application domain. This requirement rules out execution times, because they depend on the end-user's hardware. The scheduler however requires execution time information to calculate a schedule and to anticipate deadline misses. We resolve this disagreement with an indirection: ATLAS places an execution time estimator between the applications and the system-level scheduler. It decouples the interface as seen by the developer from the lower-level interface of the scheduler. Figure 3 illustrates the resulting architecture.

Both interfaces are job-based and use deadlines to express timeliness requirements. However, they differ in the way resource requirements are provided: The system scheduler expects execution times or estimates thereof. The application however provides workload metrics to the estimator, which then uses machine learning to derive execution time predictions. We now describe the task model and the interface primitives.

A. Task Model

ATLAS runs a finite set $T = \{\tau_i \mid i = 1, \ldots, n\}$ of tasks τ_i. The task set is dynamic, because applications are started and stopped at the user's discretion. Each task τ_i is a set $\tau_i = \{J_{i,j} \mid j = 1, \ldots, m\}$ of aperiodic jobs $J_{i,j}$. While theoretically unbounded, at any given time the number of jobs per task is finite, because we do not use a periodic tasks model with infinite sets of jobs. Within this paper, we use i as the task number and j as the job number within the task. At the system scheduler level, each job is described as a pair $J_{i,j} = (e_{i,j}, d_{i,j})$ of an execution time $e_{i,j}$ and an absolute deadline $d_{i,j}$. Applications can release jobs whenever they want, no minimum inter-release time and consequently no maximum demand can be assumed. This design decision was made to simplify application development, but it also prevents feasibility analysis. Applications can certainly drive the system into overload situations.

While the deadlines $d_{i,j}$ are provided by the application directly, the execution times $e_{i,j}$ are not. The application provides its estimator with job descriptions $J'_{i,j} = (\underline{m}_{i,j}, d_{i,j})$ consisting of a vector $\underline{m}_{i,j}$ of workload metrics and the same deadline as seen by the system scheduler. From the metrics, the ATLAS estimator derives estimations for the actual execution times. It can provide these estimates before the job executes. Workload metrics are positive real numbers describing the computational weight of the workload. The estimator expects at least a subset of them to correlate with the execution time. More details on the nature of those metrics and examples follow in Section IV.

B. Scheduling Primitives

This estimator is implemented as a library that is linked against applications. It collects job descriptions in one job queue per task and predicts the per-job execution times. The estimator trains itself using the application-provided metrics and a history of measured actual execution times of past jobs. With the predicted execution times, the estimator for each of its jobs $J'_{i,j}$ forwards a translated job $J_{i,j}$ to the system scheduler. The scheduler maintains a global queue of all tasks in the system. To perform the above operations, the

estimator exposes the following three interface primitives to applications:

1) submit(i, $J'_{i,j}$): As soon as an application learns about a job it needs to perform, it should register the job with the estimator. The application passes the number i of the task this job belongs to and a job description $J'_{i,j} = \left(\underline{m}_{i,j}, d_{i,j}\right)$ that includes a deadline $d_{i,j}$ and a vector $\underline{m}_{i,j}$ of workload metrics. When a job is submitted, the estimator uses $\underline{m}_{i,j}$ and its internal training state to predict the job's execution time $e_{i,j}$ and forwards the resulting job $J_{i,j} = (e_{i,j}, d_{i,j})$ to the system scheduler. It keeps the job $J'_{i,j}$ in its local queue to automatically train the estimation once the actual execution time of the job is known.

Applications can submit jobs long before actual execution begins and should do so as early as possible to allow the scheduler to look into the future. The promise an application makes by submitting a job $J'_{i,j}$ is the following: As soon as the work of all previously submitted jobs $J'_{i,k<j}$ within task τ_i is finished, the application will, when given CPU time, start working on the submitted job $J'_{i,j}$. In other words: there is no extra work performed between $J'_{i,j-1}$ and $J'_{i,j}$ that is not covered by the submitted jobs. This requirement implies that applications execute jobs in submission order. This way, the scheduler is aware of all work the applications perform within a task τ_i.

2) next(i): Because jobs are required to execute back to back within a task, this primitive notifies the estimator that both the current job within task τ_i is finished and that the next job should start. Because the estimator tracks the actual execution times of finished jobs, it finalizes the time measurement of the finished job and starts the time measurement of the new job. It also informs the system scheduler of the job switch.

3) cancel(i, j): Applications can use the cancel primitive to revoke a previously submitted job $J_{i,j}$ of task τ_i. We think this is not a common operation, but we list it here for completeness. A video player would use cancel when the user stops playback midway in the video. In the following, we will ignore this primitive because we think it yields no additional insights.

C. System Scheduler

Any scheduling algorithm compatible with the presented task model of aperiodic jobs $J_{i,j} = (e_{i,j}, d_{i,j})$ is suitable as the system scheduler. We implemented an ATLAS system scheduler in Linux 3.5.4. The patch adds an additional scheduling layer between the existing CFS and the priority-based POSIX real-time scheduling bands. Two new system calls implement the interface between the ATLAS estimator and the system scheduler.

1) atlas_submit: The `atlas_submit` system call informs the scheduler about a new job. It is used by the *submit* primitive to pass the estimated execution time and the application-provided deadline to the kernel. The kernel also receives the thread ID of the submission target. This information is necessary, because a thread can not only submit new jobs for itself, but also for another thread. This is useful for producer-consumer scenarios, where the producer thread will submit jobs for the consumer thread.

2) atlas_next: This system call is used by the *next* primitive to inform the scheduler that the thread is now ready to begin the next job. No parameters are necessary, because `atlas_next` always affects the calling thread. If the system scheduler decides to run a different thread instead, the `atlas_next` system call blocks until the scheduler picks the caller again.

The ATLAS scheduler in the kernel remembers submitted jobs in deadline order and uses the execution time estimates to maintain a LRT (Latest Release Time) schedule[1] of all jobs in the system. The slack time of this LRT schedule is managed by the default CFS scheduler, providing backwards compatibility for non-ATLAS and best-effort work. Because LRT moves slack to the front of the schedule, CFS tasks receive good response times. Whenever a release instant in the LRT schedule is reached, the ATLAS layer preempts any work running in CFS and instead switches to the LRT job. Unlike the traditional Linux real-time scheduling classes, ATLAS scheduling is not limited to the root user. The scheduler demotes misbehaving applications to CFS so they cannot seize CPU time beyond their reservation.

IV. WORKLOAD METRICS

Using workload metrics from the application domain instead of calling for developers to provide actual execution times directly is at the heart of this work. Recall that applications provide the metrics as part of each job description $J'_{i,j}$ in a vector $\underline{m}_{i,j}$ of real numbers to their individual estimator. From the estimator's point of view, each element of the metrics vector is assumed to have a positive linear correlation with the job execution time. Badly correlated elements will be filtered out to improve numerical stability, so the developer can select metrics on a "larger value means more work" basis. Expected iteration counts of loops or the expected number of times an expensive function is called are good candidates. Metrics vectors of different jobs within the same task must be compatible, that is, the same metric must be delivered in the same vector element.

A. Auto-Training Execution Time Prediction

The method to produce execution time estimates from such metrics is based on previous work [1], which has been modified to remove the offline training phase. The prediction state is maintained individually for each real-time task τ_i.

[1]Latest Release Time is essentially EDF backwards. All jobs start at the latest possible instant where they can still meet all deadlines. Like EDF, LRT is optimal regarding schedulability on a single processor with independent jobs. LRT is often intuitively used by humans when working towards a deadline: estimate the execution time and start as late as possible. From personal experience, this also applies to writing scientific papers.

However, for clarity, we will omit the i-index on each variable below.

The estimator conceptually collects a history of metrics vectors in a *metrics matrix* M, where each individual vector $\underline{m}_j$ contributes a row of the matrix. For jobs already executed, the estimator also knows the actual, measured execution time of the jobs and collects them in a column vector $\underline{t}$. To predict execution times of future jobs, the estimator uses this history to obtain a coefficient vector $\underline{x}$ that approximates the execution times when applied to past metrics:

$$M\underline{x} \approx \underline{t}$$

The number of metrics per job is expected to be small compared to the number of jobs, so the above linear system has more rows than columns. The equation is therefore overdetermined, so we reformulate it as a linear least square problem:

$$\|M\underline{x} - \underline{t}\|_2 \to \min_{\underline{x}}$$

Given such coefficients $\underline{x}$ and a metrics vector $\underline{m}$ of a new job description J', the estimator can calculate the execution time prediction as a dot product of both:

$$e = \underline{m} \cdot \underline{x}$$

This estimated time is forwarded to the system scheduler.

The coefficient vector $\underline{x}$ should be determined without an offline training phase. A naive implementation would continuously collect metrics and measured execution times and from time to time solve the linear least square problem to obtain an updated coefficient vector $\underline{x}$. As jobs are executed, new metrics vectors and measured execution times add more rows to the problem, so the computation required to update $\underline{x}$ would increase over time. Literature on numerical algorithms however offers self-updating solutions, preventing the unbounded row growth: Given an upper triangular matrix $R^{(k)}$ solving[2] a linear least square problem $\left(M^{(k)}, \underline{t}^{(k)}\right)$, a new job being executed adds a new row to both the metrics matrix and the vector of measured execution times. The resulting new problem is $\left(M^{(k+1)}, \underline{t}^{(k+1)}\right)$. The matrix $R^{(k+1)}$ which solves this new problem can be calculated from $R^{(k)}$ and the newly added row alone, without the need for the entire $M^{(k+1)}$ or $\underline{t}^{(k+1)}$. The new row is appended to $R^{(k)}$ and Givens rotations are applied to $R^{(k)}$ to turn it into upper triangular form again. The coefficients $\underline{x}$ can be determined easily from the resulting $R^{(k+1)}$. With this *updating predictor*, only the current value of the constant-size R needs to be stored, the ever-growing matrix M and vector $\underline{t}$ need not be stored, because new rows are directly merged into R.

This updating predictor always provides a coefficient vector $\underline{x}$ that considers all past jobs. Due to numerical problems like rounding errors, the solution can get stuck at a global average and not adapt to new behavior any more. We compensated for this by including an *aging factor* in the updating predictor. This factor slightly downscales the influence of past knowledge and allows the solution to remain flexible and lean towards newly acquired knowledge. Because any past behavior of the task may resurface, we argue that this aging should be small and only allow for slow adaptation by a long-term sliding average. Therefore, we chose an aging factor of 0.999, which reduces the weight of past knowledge to 10% after 2300 jobs.

[2]A matrix R is said to solve a linear least square problem $(M, \underline{t})$, if $QR = M$ with Q being an orthogonal and R being an upper triangular matrix. The solution $\underline{x}$ is determined by $R\underline{x} = \underline{t}$.

B. Improving Numerical Stability

A problem we already encountered in previous work [1] is instability due to over-fitting: When the linear least square problem is solved for a workload that does not sufficiently exercise all metrics, the solution might be very sensitive to small variations in some of the metrics. This instability happens primarily when the predictor is still young. We apply a similar column dropping technique to ignore metrics with high potential to skew the result, but little contribution to the prediction accuracy. Thus, stability is improved at the cost of some accuracy [1]. The column dropping is performed with our updating predictor by applying appropriate Givens rotations to R.

C. Practical Considerations

1) Time Measurement: To perform the time measurement needed for the online training, the estimator needs access to a per-thread clock that only ticks when the thread is actively using the CPU and that pauses once the thread is descheduled by the system scheduler. Linux provides such a facility with the `clock_gettime(CLOCK_THREAD_CPUTIME_ID, ...)` system call. However, no per-thread clock compensated for CPU frequency changes is available. Thus, we disable frequency scaling in all our experiments as it would perturb the decoding time predictions.

2) Misprediction: The job execution time estimates $e = \underline{m} \cdot \underline{x}$ can be wrong in both directions, so we have to account for over- and underestimation. The former will just result in an early `atlas_next` call, because the thread reached the end of the job sooner than expected. The ATLAS system scheduler passes the additional slack to CFS. Underestimation is a concern, because a longer execution time will lead to missed deadlines due to the nature of the LRT plan.

We remedy this problem with three measures. First, the ATLAS job at the front of the LRT plan is also schedulable by CFS, allowing it to run early, if CFS decides to give it CPU time. This frontmost ATLAS job is therefore released according to EDF, but competes with all CFS threads for the CPU. This head-start is not accounted against the estimated execution time. As soon as the current job's latest release instant is reached, it is scheduled exclusively, shutting out

the CFS work. The resulting scheduling behavior is a hybrid between EDF and LRT.

Second, we over-allocate all jobs by 1% when they are inserted in the kernel's LRT plan. This over-allocation provides robustness against rounding errors in the system scheduler's timekeeping and small underestimations. Because more time is assigned to each job, the system scheduler will observe a higher utilization, resulting in a schedulability loss of the same 1%.

Third, as a last resort, all jobs overrunning their estimated execution time plus the over-allocation will be demoted to CFS and are only scheduled in the slack of the LRT plan to prevent interference with honest jobs. Demoted jobs recover, when they signal job completion using the `atlas_next` system call. LRT moves slack to the front of the schedule, allowing late jobs to catch up as soon as possible.

3) Overhead: Every newly submitted job must be inserted into the scheduler's LRT plan. This operation has a worst-case complexity of $O(n)$, because the new job can potentially cause all other jobs to move. Application developers should understand that job submission can pose a non-negligible load on the system scheduler. Too fine-grained packaging of their real-time load into jobs should thus be avoided. Modeling individual frames as jobs for a video player is an adequate choice, typically causing less than 100 job submissions per second.

4) Initialization: Because the estimator trains itself while it is running, the prediction accuracy can be diminished at the beginning of a real-time task. This can be avoided by saving the running solution matrix R when the application terminates, so a later start of the same software can build on this knowledge. To generate an initial solution, a short training load could be executed during installation of the application.

V. Evaluation

We implemented the ATLAS estimator as a C library and the ATLAS system scheduler as a Linux patch against kernel version 3.5.4. All measurements and experiments[3] have been conducted on a 2.4 GHz Intel Core i5 with 4 GB of 1067 MHz DDR3 RAM. The relevant threads are pinned to the first processor core, because our ATLAS implementation does not spread work across cores yet.

To evaluate ATLAS behavior with a complex real-time application, we modified the FFplay video player from the FFmpeg [3] project (version 0.11.1, released in June 2012) to use the ATLAS interface. FFplay is a lean player with a spartan user interface, but otherwise fully featured. Because multicore operation and overload mitigation are not evaluated here, we always run FFplay with the `-threads 1` and `-noframedrop` command line options.

[3]All software to reproduce the experiments in this paper can be obtained from our public subversion repository: https://os.inf.tu-dresden.de/~mroi/svn/video/branches/2013-RTAS-ATLAS/

```
void input_thread() {
  for (;;) {
    frame = read_frame();
    metrics = extract_metrics(frame);
    deadline = timer +
      video_queue.size * frame_time;
    atlas_job_submit(decoder_thread,
      deadline, metrics);
    video_queue.put(frame);
  }
}

void decoder_thread() {
  for (;;) {
    atlas_job_next();
    frame = video_queue.get();
    decode(frame);
    picture_queue.put(frame);
  }
}
```

Figure 4. Simplified FFplay Excerpt Showing ATLAS Integration

We focus on the H.264 video decoding standard [2] as the video workload. It is widely deployed and used in the majority of modern media applications ranging from mobile videos to HD and 3D Blu-ray discs. We used the videos listed in Table I. The list contains short test clips to demonstrate that the estimation quickly produces reasonable results as well as two movie trailers to show the long-term stability. All videos but one have been encoded by us with the x264 encoder [7] (Git revision 37be5521, dated June 2012). The Shore video was encoded by the manufacturer of the commercial FastVDO encoder. Audio is ignored, because its computational load is two orders of magnitude smaller than video decoding and its timing requirements are less strict, because sample buffering and timely output are managed by the audio hardware.

We answer the following questions in the evaluation: What changes are necessary to use ATLAS scheduling in FFplay? How well can workload metrics capture the varying execution time of the decoder stage? How does the choice of metrics affect scheduling behavior? And finally, what is the overhead of ATLAS?

A. FFplay Integration

Three worker threads are involved in video playback and all are managed by ATLAS. The input thread is I/O-bound, the decoder thread CPU intensive and the display refresh thread is self-suspending, because it waits for the right time to display the next frame. Queues connect the stages, resulting in producer-consumer dependencies.

Because ATLAS can submit jobs across thread boundaries, the interface primitives map well to the given player architecture. Whenever the input thread enqueues a compressed frame, it knows the decoder will invest CPU time to decode it. Consequently, this position in the code is the natural

Table I
TEST VIDEOS USED FOR THE EVALUATION

Video Name	Content	Resolution	FPS	Duration	Profile	Encoding
Shore [4]	flight over a shoreline at dawn	740×576	25	27 s	High	FastVDO
Black Swan [5]	movie trailer for "Black Swan" (2010)	848×352	24	126 s	Baseline	x264, CBR, 1500 kbit/s
Parkrun [6]	man running in a park with trees, snow and water	1280×720	50	10 s	High	x264, CBR, 4 Mbit/s
Rush Hour [6]	rush-hour in downtown Munich with heat haze	1920×1080	25	20 s	Main	x264, CRF, ratefactor 23
Hunger Games [5]	movie trailer for "The Hunger Games" (2012)	1920×816	24	155 s	High	x264, CRF, ratefactor 23

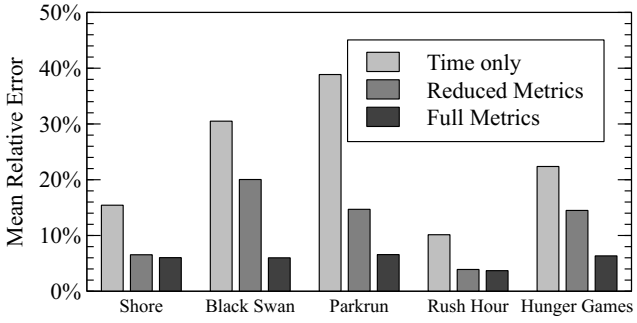

Figure 5. Comparing Prediction Accuracy for Different Metrics

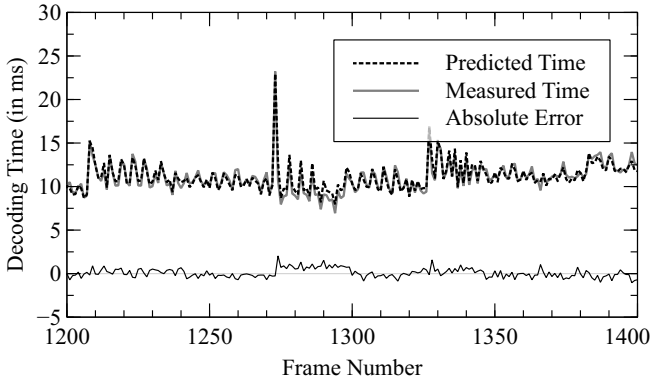

Figure 6. Prediction Detail for "Hunger Games"

place to submit a job to the decoder thread. The input thread needs to include a deadline and workload metrics in the job submission. The compressed frame influences how long it will take the decoder to finish the frame. The input thread therefore inspects the compressed frame and extracts workload metrics from it. We will consider different options for those metrics below. The deadline is determined by the frame rate of the video multiplied with the number of yet to be decoded frames in the queue. This way, the input thread submits the job and enqueues the frame for the decoder, which simply informs the scheduler about its progress with the *next* primitive. Figure 4 illustrates the use of the scheduling primitives. We instrumented all FFplay threads in this way.

Three points should be highlighted: First, the modifications are small. Adding two function calls, the deadline calculation and the extraction of the workload metrics is all that is needed. The ATLAS estimator is linked to FFplay as a library. Second, job submission and execution can happen in different threads, as it does in our FFplay example. For such producer-consumer-scenarios, the use of the ATLAS primitives naturally follows the logical data flow in the program. Third, all data we ask of the developer is close to the application's problem domain. Deadlines are dictated by the video and queue state, bitstream metrics are used to describe the workload. We therefore think, the presented scheduler infrastructure is easy to use for developers.

B. Options for Workload Metrics

Because it is the dominant CPU task and varies most in execution time, we now focus on the decoder thread as the most challenging to schedule. We evaluate three variants of the workload metrics specifying the decode jobs. The first and easiest one is to assume constant execution time by passing no metrics. The resulting estimates will be a sliding average over the previously observed execution times, which is the baseline any non-workload-aware reservation will achieve. The second option is to use metrics that can be observed from the bitstream very easily:

- number of pixels of the frame
- bytes of the compressed frame
- frame type (I, P, or B)

The first two values are used as metrics directly, the frame type is passed to the estimator as three separate metrics, each of which is either 0 or 1. So we use $(1, 0, 0)$ for an I-frame, $(0, 1, 0)$ for a P- and $(0, 0, 1)$ for a B-frame. We call this option "reduced metrics".

The best estimation results however are obtained when using the full decoding time metrics we identified in previous work [8]. Most of these metrics however are hidden underneath an outer layer of entropy coding, so some decoding work needs to be performed to reveal those metrics. This can be performed either offline or online. We chose an offline preprocessing step similar to what we presented earlier [9]. During preprocessing, the video metrics are embedded into the video stream as user-defined datagrams. MPEG-4 pt. 2 video can contain similar information in the complexity estimation header, we use a similar approach for H.264. An online solution would employ a staged decoder as suggested in [10]: The first stage decodes the entropy compression

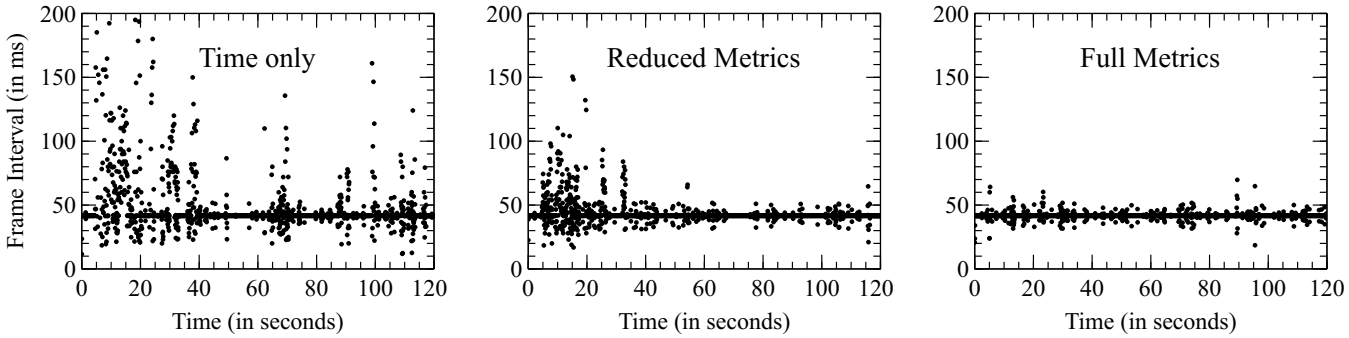

Figure 7. Frame Jitter with Background Load

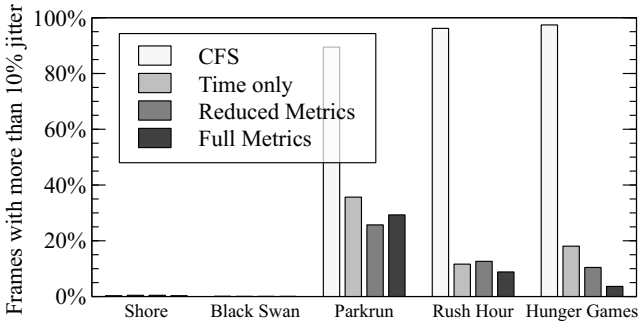

Figure 8. Video Quality with Different Metric Choices

layer to extract the metrics for the second stage, which performs the remaining decoding steps. This would obsolete the preprocessing entirely, but requires more intricate changes to the video decoder. If neither of these approaches is desired, the reduced metrics must be used.

Figure 5 shows that the metrics improve prediction accuracy over a purely time-based predictor. The presented mean error was computed by averaging the relative errors, which are the absolute difference of predicted and measured values, normalized to the measured value. The numbers show that a non-workload-aware predictor has fairly high relative errors. The reduced metrics that do not require preprocessing or a staged decoder provide a good middle ground, whereas the full set of metrics offers very good prediction accuracy, with typical relative errors below 10%. Figure 6 zooms in on a section of the Hunger Games video to illustrate that the full-metrics prediction tightly follows the large variations of the decoding time.

C. Scheduling Behavior

What matters to the user in the end is the overall scheduling behavior. We ran all videos against a background load of ten CPU-hogging processes and judged video quality by the fraction of frames with a display jitter greater than 10% of the nominal frame interval. Figure 8 shows the impact of the three metric choices outlined above and compares with Linux' standard CFS scheduler. The Shore and Black Swan videos are standard definition videos and are successfully scheduled even by CFS with the given background load. The three following videos are high definition and as demonstrated in Figure 1, their quality deteriorates when scheduled by CFS. All metric choices lead to acceptable behavior and improve the situation compared to CFS. Interestingly, the Parkrun and Rush Hour videos occasionally exhibit better behavior with less precise metrics. Because both videos consist only of a single scene, the decoding times do not fluctuate much and because the videos start with a time-intensive I-frame, the moving-average based metrics tend to over-approximate the execution times, thus leading to better scheduling behavior. The Hunger Games trailer shows the advantage of the full metrics, but the reduced metrics and time-based prediction are also competitive. Figure 7 illustrates the scheduling behavior for the Hunger Games video, showing the differences between the metrics along the time axis.

D. Scheduling Overhead

The runtime overhead added by the ATLAS estimator and scheduler is below 1% in the conducted experiments. We want to point out that the predictions are available before the decoding job executes. FFplay's video queue between input and decoder thread determines how far ahead of the decoding the estimation is performed. In its standard configuration the queue holds 15 MiB of compressed data or 5 compressed frames, whichever is reached earlier. The execution time estimates presented above are therefore typically[4] available 5 frames ahead of time.

VI. Related Work

We look at scheduler architectures and interfaces within a spectrum ranging from fair-share schedulers to periodic task models. Fair-share approaches are found in today's commodity operating systems and offer very little to no chance for applications to expose timeliness requirements. Consequently, they are easy to program against, but offer weak guarantees with respect to timing. Strong guarantees

[4]With a highly compressed format like H.264, the frame limit is almost always reached before the size limit.

are obtained from schedulers for periodic tasks, because the assumption of periodic behavior gives the scheduler implicit clairvoyance into the applications' future and allows formal task admission. However, these systems are more difficult to program because the developer must fit his application into the periodic model and provide correct task parameters such as worst-case execution times.

A. Reservation Approaches

One influential approach to tame periodic tasks is the Constant Bandwidth Server (CBS) [11]. It 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 by Abeni, Palopoli, et al. added dynamic changes to the allocated server bandwidth [12] and slack reclaiming mechanisms [13]. CBS-based schedulers have also been proven to operate well on multicore systems [14] and within frameworks for quality of service control [15], both are areas of future work for ATLAS.

At its 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. ATLAS 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 ATLAS is the use of a per-task execution time estimator. However, the estimator presented for CBS [12] extrapolates by using only past execution times. ATLAS has shown that workload metrics improve estimations, so an adaptive reservation system using the ATLAS estimator could provide an interesting QoS-aware overload mitigation mechanism.

B. Lightweight Specification

A further extension to CBS allows it to determine all relevant task parameters automatically [16]. This brings CBS all the way to the other end of our spectrum, turning it into a heuristical scheduler with no interface. However, applications know about their timing requirements and it should not be difficult for programmers to expose them. We have shown that ATLAS requires little modification to a complex real-time application to forward deadlines and workload metrics to the scheduler. This saves the elaborate black-box observation to automatically determine task parameters.

Redline [17] is a comprehensive approach to scheduling that needs no modification of applications, but relies on externally provided specifications. The Redline example specification for mplayer, a Linux video player, reserves 5ms of CPU time every 30ms. Even though Redline will dynamically adapt the actual CPU budget at runtime, this static specification does not adequately address the high variability of decoder load. The metrics used by the ATLAS estimator automatically provide workload-aware task parameters. Apart from CPU, Redline also manages disk and memory, while ATLAS deals with CPU only.

Another scheduling infrastructure that targets high-throughput applications with timing constraints is Cooperative Polling [18]. Krasic et al. share our view that applications should be modified to expose their timing needs to the scheduler, but the model needs to be simple for application developers. Scheduling is split between an application scheduler and the kernel and the interface primitives show similarities to ATLAS. Cooperative polling however lacks a facility to provide execution times to the scheduler. CPU usage is decided by the scheduler according to a fairness policy. ATLAS adds auto-trained execution time estimates that allow early detection of future deadline misses.

C. Fair Processor Sharing

On the other end of the spectrum, fair-share schedulers allow limited control over scheduling behavior by providing a priority interface like the Unix nice levels. While fixed per-task priority assignments are sufficient to enable rate monotonic scheduling, priorities require global knowledge. A developer cannot determine the correct priority level for an application without a complete overview of all other applications in the system. Extensions to the fair-share paradigm like Borrowed Virtual Time (BVT) [19] express task priorities with a time — called warp time with BVT — but assigning these parameters still requires global system knowledge, because warp times are not intrinsic to one application. The largest warp time in the system determines task dispatch just like priorities. ATLAS 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.

After this brief survey of related work, we believe the ideas of exploiting buffering for look-ahead and providing workload metrics to allow execution time prediction are unique to ATLAS. Compared to approaches based on runtime measurements or profiling, ATLAS provides accurate execution time estimates for a job before it is executed. ATLAS currently lacks multicore support and overload mitigation, both of which we will discuss in the following section.

VII. OUTLOOK

Driven by the pervasiveness of multicore systems, we see an interesting challenge in combining real-time and parallel programming. In order to reliably detect overload situations, ATLAS must differentiate jobs that can only execute serially on a single CPU from jobs that can be spread across multiple

CPUs. We need to find a way to inform the scheduler of the parallel nature of the jobs.

We plan to run the presented ATLAS LRT scheduler as a per-core scheduler. An overarching load balancer migrates work between cores if it detects a need to do so. This design is similar to the default CFS scheduler in Linux. But while CFS can only determine the current load heuristically, ATLAS offers more precise knowledge about the current and future load on the cores. We want to explore using this knowledge to drive interesting multicore work balancing strategies.

Since ATLAS does not support a classical admission, the system's reaction to a detected overload also poses an interesting research challenge. We currently contemplate a back-pressure system in which the system scheduler would issue an upcall to applications, asking them to reduce their load. A video player has some opportunities, depending on the video being played, to lower the quality and thereby reduce CPU demand. ATLAS detects overload situations before they occur, allowing applications to prepare for the overload. A video player can adapt by degrading any frame to be decoded before the anticipated deadline miss. We have demonstrated earlier [9] that more degradation options allow the player to make quality-aware decisions that lead to better visual quality for the user.

VIII. Conclusion

We presented ATLAS, the auto-training look-ahead scheduler. It offers a simple interface to application developers, only asking for parameters from the problem domain of the application. Explicit job submission replaces the periodic task model. Timing constraints are exclusively expressed with deadlines. Workload metrics are used instead of execution times, because those times are hardware- and workload-dependent and therefore hard to obtain for developers. An estimator uses the metrics to deliver accurate predictions for execution times of future jobs. The predictor does not require a separate training process and provides typical relative errors below 10%. ATLAS features look-ahead, because execution time predictions are available ahead-of-time, which allows detecting overload situations before they occur.

Acknowledgments: We thank our colleagues for supporting the work and proof-reading the paper. We express our gratitude to the anonymous reviewers and to our shepherd Gernot Heiser, whose guidance helped improving the paper. We also want to recognize the HAEC project for inspiring and funding this research effort.

References

[1] M. Roitzsch and M. Pohlack, "Principles for the prediction of video decoding times applied to MPEG-1/2 and MPEG-4 part 2 video," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006, pp. 271–280.

[2] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[3] "FFmpeg project." [Online]. Available: http://ffmpeg.org/

[4] "FastVDO test videos." [Online]. Available: http://www.fastvdo.com/H.264.html

[5] "iTunes movie trailers." [Online]. Available: http://trailers.apple.com/

[6] "High-definition test sequences." [Online]. Available: http://media.xiph.org/video/derf/

[7] "x264 project." [Online]. Available: http://www.videolan.org/developers/x264.html

[8] M. Roitzsch, "Slice-balancing H.264 video encoding for improved scalability of multicore decoding," in *Proceedings of the 7th International Conference on Embedded Software (EMSOFT)*, October 2007, pp. 269–278.

[9] M. Roitzsch and M. Pohlack, "Video quality and system resources: Scheduling two opponents," *Journal of Visual Communication and Image Representation*, vol. 19, no. 8, pp. 473–488, December 2008.

[10] P. Altenbernd, L.-O. Burchard, and F. Stappert, "Worst-case execution times analysis of mpeg-2 decoding," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*, June 2000, pp. 73–80.

[11] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, December 1998, pp. 4–13.

[12] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, "QoS management through adaptive reservations," *Real-Time Systems*, vol. 29, pp. 131–155, 2005.

[13] L. Palopoli, L. Abeni, T. Cucinotta, G. Lipari, and S. K. Baruah, "Weighted feedback reclaiming for multimedia applications," in *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*, October 2008, pp. 121–126.

[14] S. Kato, R. Rajkumar, and Y. Ishikawa, "AIRS: Supporting interactive real-time applications on multicore platforms," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010, pp. 47–56.

[15] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari, "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, November 2010.

[16] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys)*, April 2010, pp. 55–68.

[17] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: First class support for interactivity in commodity operating systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, December 2008, pp. 73–86.

[18] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel, "Fair and timely scheduling via cooperative polling," in *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*, April 2009, pp. 103–116.

[19] 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 (SOSP)*, December 1999, pp. 261–276.