Concepts for Execution Time Prediction of 3D GPU Rendering

Stephan Schnitzer*, Simon Gansel†,
Frank Dürr* and Kurt Rothermel*
*Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
Email: lastname <at> ipvs.uni-stuttgart.de
†System Architecture and Platforms Department, Mercedes-Benz Cars Division, Daimler AG, Germany
Email: firstname.lastname <at> daimler.com

Abstract—**The relevance of graphical functions in vehicular applications has increased significantly during the last years. Modern cars are equipped with multiple displays used by different applications such as speedometer, navigation system, or media players. The recent trend towards hardware consolidation to reduce hardware cost, installation space, and energy consumption, causes graphical 3D applications of different safety-criticality to share a single GPU. This requires effective real-time GPU scheduling concepts to ensure safety and isolation for 3D rendering. Since current GPUs are not preemptible, a deadline-based scheduler must know the GPU execution time of GPU commands in advance. In this work, we present a novel framework to measure and predict the execution time of GPU commands using OpenGL ES 2.0. We present prediction models for the main GPU commands relevant for 3D rendering, namely, FLUSH, CLEAR, and DRAW. For the DRAW command we propose to use the 3D bounding box of the rendered model and apply the vertex shader projection to heuristically estimate the number of fragments rendered. We finally present the implementation and evaluation of our framework, which demonstrates its feasibility and shows that good prediction accuracy can be achieved. In our evaluation using realistic scenarios the absolute prediction error of the DRAW command did not exceed $260\,\mu$s.**

Keywords*—*GPU-scheduling, 3D-rendering, real-time, embedded systems, execution time prediction

I. INTRODUCTION

Innovations in cars are mainly driven by electronics and software today [1]. In particular, graphical functions and applications enjoy growing popularity as shown by the increasing number of displays integrated into cars. For instance, the head unit (HU) uses the center console screen to display the navigation system, or displays integrated into the headrests of the front seats to display multimedia content. Another recent trend in modern cars is to replace the analog instruments of the instrument cluster (IC) by digital displays, for instance, for displaying speed information or warning messages.

Although in the beginning graphical output was mainly 2D content like movies or 2D maps, the amount of 3D graphics is steadily increasing [2, 3]. For instance, modern navigation systems display 3D city models, the instruments of the vehicle are rendered 3D objects with reflections and shadows to imitate physical instruments as close as possible, or a "bird's eye view" with a virtual 3D model of the car and its surrounding assists the driver during parking. To render such complex scenes with high frame rates graphical processing units (GPUs) are integrated into cars. Traditionally, each system like the HU or IC uses dedicated GPUs for rendering. However, multiple GPUs increase cost, energy consumption, and space requirements. Therefore, there is a strong incentive to consolidate hardware, and ultimately share a single GPU between several applications like displaying instruments [2], navigation system, parking assistant, third-party applications from an app store [4–6], etc.

Similar to hardware sharing in classic domains such as server virtualization, sharing hardware between several applications in a car requires isolation properties to avoid interference between applications. However, in contrast to classic domains the automotive domain has very stringent safety requirements due the fact that failures might lead to substantial financial loss, e.g., due to the damage of the vehicle, or even injuries of passengers and other traffic participants. These safety requirements also involve the sharing of GPUs. For instance, it must be guaranteed that a warning message like a warning about insufficient oil pressure or a too short distance to the car ahead is displayed in time (seconds to sub-seconds) to react properly. The safety aspect of displaying information in vehicles is considered by different standards. For instance, an ISO standard [7, ISO 26262] regulates that the level of safety-criticality of each functionality has to be assessed and suitable methods must be implemented to minimize the risks caused by malfunctions. Concerning graphical representations that use safety-critical 3D rendering, it must be ensured that they are displayed or updated within certain time limits. Besides safety-criticality, updating displays in real-time is also important for usability and aesthetic reasons. For instance, unsteady updates of pointers of instruments are deemed to be unacceptable.

Non-technical approaches to guarantee real-time graphical output such as certification of software by a central authority like the OEM are not scalable since many apps are not implemented by the OEM himself but sub-contractors or even a large number of untrusted third-party developers of an app store. Therefore, technical concepts for GPU sharing with real-time guarantees are required, in particular, GPU scheduling [8]. However, the prevalent application domains for GPUs like gaming or general purpose computing on GPUs (GPGPU) did not require and therefore did not implement sufficient isolation and scheduling concepts since typically one application has exclusive access to the hardware. Therefore, the essential concept of preemption which could ease scheduling significantly is still not available for GPUs.

Consequently, we focus on an alternative scheduling concept in this paper, namely, deadline scheduling with execution time prediction at runtime, which was first proposed for GPUs by Bautin et al. [9]. The basic idea is to predict the execution time of non-interruptible batches of GPU commands—so-called command groups—, before they are sent for execution to the GPU. Command groups that would prevent the command group of a high-priority application from finishing in time are not admitted to access GPU resources.

Obviously, for this mechanism to be effective, a good prediction of the execution time of command groups is required. Kato described a history-based prediction algorithm [10] that measures the execution time of command groups during runtime and can then predict the execution time for the next execution. However, this is only effective for exactly the same command group in the exactly same context, e.g., same rendering scene. Moreover, it is also ineffective for the first execution of a command group.

Due to these significant limitations, we propose a different prediction concept for real-time GPU scheduling in this paper. The basic idea is to predict the individual execution time of graphics commands using models that are calibrated during runtime. In particular, we propose models for the main three commands relevant for 3D rendering, namely, FLUSH, CLEAR, and DRAW, using the Open Graphics Library for Embedded Systems (OpenGL ES) standard [11]. FLUSH has constant execution time independent of the context. The execution time of CLEAR essentially depends on the render buffer size. The DRAW model is based on the number of vertices and the number of fragments (possible pixels of triangles). Therefore, to predict the DRAW execution time, we need a concept to estimate the number of fragments and time to shade the vertices of the fragments using the given shader program. We achieve this by emulating the vertex shader on a bounding box of the 3D model in constant time. To calibrate the execution time of these commands on the specific GPU and to execute the emulation, we instrument the GPU command groups in kernel space on the fly.

In summary we make the following contributions: 1) A system architecture and framework for predicting the execution time of GPU commands and calibrating the underlying models through runtime measurements. 2) Models of the GPU execution time for the three OpenGL ES 2.0 commands FLUSH, CLEAR, and DRAW. 3) An implementation of the framework and the proposed prediction model. 4) An evaluation showing the accuracy of measurements and of the predicted execution time. In our evaluations the absolute prediction error of CLEAR and DRAW commands never exceeded $260\,\mu$s, supporting our claim that our heuristic is good.

The rest of this paper is structured as follows. In Sec. II we discuss related work. In Sec. III we present our system model and explain our concept in Sec. IV. In Sec. V we explain our implementation and evaluate it in Sec. VI. We conclude with a summary and an outlook on future work in Sec. VII.

II. RELATED WORK

Real-time GPU scheduling as targeted by our approach and classic real-time scheduling for CPUs [12–14] share a common goal, namely, to guarantee the timely execution of code. However, the underlying system model is fundamentally different since CPU scheduling can be based either on preemption or a known execution time of commands. Preemption mechanisms are not available for GPUs so far. Although the Windows Display Driver Model (WDDM), which was introduced with Windows Vista, supports GPU preemption since version 1.3 [15], preemption is an optional feature and not supported by current drivers. Moreover, the model does not provide any guaranteed maximum delay between preemption request and completion. Therefore, WDDM is currently not sufficient to achieve real-time GPU scheduling, leaving GPU execution time prediction indispensable.

To enable real-time GPU scheduling, prediction mechanisms for GPU command groups were proposed in several works. Bautin et al. [9] designed a system for GPU multitasking including a priority-based scheduler, called Graphics Engine Resource Manager (GERM). To this end, GERM collects statistics of the execution time of GPU command groups to calculate an average execution time per command group. However, in contrast to our approach, prediction does not consider the OpenGL context nor the actual set of GPU commands of individual command groups resulting in inaccurate predictions and violations of real-time constraints.

Kato et al. present another real-time GPU scheduler called TimeGraph [10] using user-defined scheduling policies. The scheduling policies assign GPU resources based on GPU execution time contingents of periodic intervals. For their a priori enforcement policy, they provide isolation of different 3D applications, given that GPU execution time prediction is accurate. For prediction, the authors propose a history-based approach that uses the recorded execution time of previously executed commands. The prediction algorithm checks whether a record for the (exactly) same command group binary code exists. In this case, the recorded execution time is used as prediction; otherwise the maximum execution time is assumed. To rely solely on history-based prediction means that unknown command groups cannot be predicted. Using the maximum observed execution time instead might lead to drastic over-estimation of the execution time. Furthermore, history-based prediction is not aware of the execution context. For instance two glClear commands may result in binary equivalent GPU instructions, although they refer to render buffers of different sizes and thus different execution times. As shown by the authors, this leads to significant prediction errors for complex dynamic scenes, which occur in many rendering applications today. Therefore, we based our prediction model on more complex models including context information.

In [16], Yu et al. propose a resource management framework called Virtualized GPU Resource Isolation and Scheduling (VGRIS) targeted at cloud gaming systems. VGRIS provides three scheduling algorithms for different kinds of GPU computation tasks, namely, Service Level Agreement (SLA)-aware scheduling, proportional-share scheduling, and hybrid scheduling that combines the former two. The SLA policy aims to provide a stable average frame latency. To this end, the computation time of the *Present* command (similar to eglSwapBuffers command in EGL), which represents the execution of one frame on the GPU, has to be predicted. Similar to Kato [10] the authors use history information about the last *Present* commands of the application. More precisely, they use the average time of the last twenty *Present* commands to predict the next *Present* command. Hence, VGRIS is actually less accurate than [10], since only fully rendered frames are measured and scheduled rather than GPU command groups (typically, a frame is rendered using multiple command groups). Furthermore, they assume that the applications are well known and have a specified rendering behavior. Therefore, an application never calling the *Present* command would be granted infinite GPU execution time, possibly blocking all other applications.

III. System Model

Before we present our technical contributions, we first introduce our system model and assumptions. For the sake of a concrete description, we consider a specific state-of-the-art graphics API, operating system, and graphic system architecture, namely, OpenGL ES 2.0 [11], Linux, and the Nvidia Nouveau architecture [17], respectively. However, the basic prediction concepts proposed in this paper are generic.

The components and interfaces of our system are depicted in Fig. 1. Basically, the system consists of components of three layers, namely, application-layer components, user-space driver components, and kernel-space components.

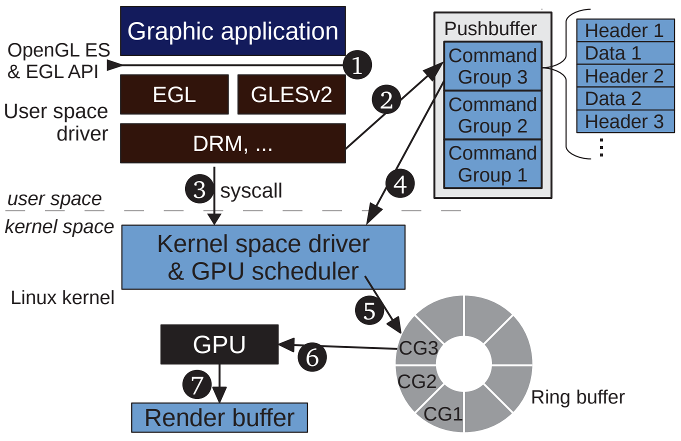

Figure 1. Nouveau pushbuffer architecture

The *graphic application* uses EGL as interface to the windowing system and the OpenGL ES 2.0 API for rendering (❶ in Fig. 1). From the OpenGL commands, the *user-space driver* creates GPU commands. We classify the relevant OpenGL commands used for 3D rendering into the following classes:

- FLUSH flushes the rendering pipeline and is caused by OpenGL calls like glFlush, glFinish, or eglSwapBuffers.
- CLEAR is caused by glClear and assigns the background color, typically at the beginning of the rendering loop.
- DRAW initiates the rendering pipeline to draw primitives using glDrawArrays or glDrawElements.

Consecutive batches of GPU commands are called *command groups* and are added to the so-called *pushbuffer*. The pushbuffer resides in shared memory accessible from user space and kernel space (❷ and ❹). Eventually, the driver flushes the pushbuffer to notify the *kernel space driver* of the added command group using a system call (❸). Coordinated by the GPU scheduler, the kernel space driver puts references to command groups into a ring buffer (❺). The GPU uses the ring buffer to fetch command groups, execute them, and update the render buffer if a DRAW or CLEAR command was included in the command group (❻ and ❼). Once the GPU starts execution, the execution of command groups cannot be preempted.

It is important to note that the execution time of OpenGL commands depends on their *context*. In particular, the execution time of the DRAW command is context-dependent. For instance, the application can set so-called shader programs with glUseProgram that are used by subsequent draw commands for geometric transformations and the definition of pixel colors. Depending on the complexity of the shader program and scene (e.g., number, size, or composition of triangles), draw commands might take different amounts of time. Therefore, in order to allow for non-preemptive real-time scheduling of command groups, their execution time must be predicted prior to their execution. To this end, we introduce a *predictor* component, which implements prediction algorithms based on context-sensitive prediction models for individual commands. The scheduler only puts a command group into the ring buffer if the predicted execution time would not violate given real-time constraints. The focus of this work is on the prediction framework and accurate prediction models and algorithms; scheduling is beyond the scope of this paper.

Next, we describe our prediction framework, prediction algorithms, and models in more detail.

IV. Prediction Framework

In this section we describe our prediction framework. We start with an overview of the architecture and then describe the mechanisms and concepts for measuring the execution time of GPU command groups and predicting selected commands.

A. Architecture

Fig. 2 shows an overview of the framework architecture with its four basic components, namely, OpenGL Context Monitor, Predictor, GPU Scheduler, and Execution Time Monitor and their embedding into the overall system. The main component

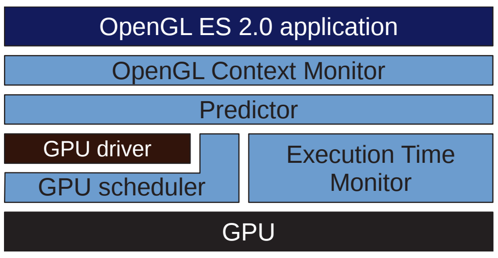

Figure 2. Framework architecture

of our framework is the *Predictor*, which predicts the execution time of command groups. To this end, it needs two sources of information:

1) The current OpenGL context, which influences the execution time of GPU commands as motivated above. All contexts are traced by the OpenGL Context Monitor.
2) Execution time models which are calibrated for the specific GPU hardware. In order to calibrate these models, we use execution time measurements measured by the Execution Time Monitor at runtime.

The Predictor predicts the execution time of command groups by predicting the execution time of each single command of the group and summing up the values. To predict the execution time of individual commands, the predictor feeds his calibrated execution time models of the commands with the current context information. The predictor is notified when calls to the OpenGL functions of the GPU driver actually emit a command group. This feature is added to the GPU driver and allows the predictor to attach the predicted execution time of a command group to the system call.

Basically, there are two places where to implement the Predictor: in user-space above the GPU driver or kernel-space below the GPU driver. We decided to implement the Predictor in user-space since context information can be determined much easier in user-space than in kernel-space. In user-space, this context can be inferred from the commands transmitted through the OpenGL API, whereas in kernel-space we would need to disassemble the binary GPU code and interpret hardware-specific GPU commands and data formats. The predicted execution time of a command group then serves as input to the *GPU Scheduler*. Since the process of GPU scheduling is out of the scope of this paper, we focus on context monitoring, the prediction models, and execution time monitoring next.

B. Context Monitoring

As described earlier, the OpenGL context impacts the execution time of rendering commands. Therefore, the OpenGL Context Monitor of our framework intercepts all relevant OpenGL and EGL API calls to create and maintain a local copy of the OpenGL state. More precisely, applications create one or more OpenGL contexts. In order to submit rendering commands, an application thread must first activate a context using eglMakeCurrent. Subsequent OpenGL calls all refer to the active context. To reflect this, we allocate a data structure for each created context. A thread-local variable holds a reference to the currently active context. The context data structure contains all relevant parameters and is updated by subsequent OpenGL calls by the application. For instance, if the application calls glBindBuffer, the Context Monitor assigns the id of the chosen buffer to the currently active context data structure, thus determining the buffer data source or target for subsequent OpenGL commands.

C. Prediction Model: FLUSH Command

The FLUSH command pushes the current command group to the kernel-space driver by using a system call. It is caused by multiple API calls, e.g., glFlush, glFinish, or eglMakeCurrent. Since the FLUSH command does not depend on parameters and has no context dependencies, its execution time is constant for a given system, i.e., it only depends on the specific speed of the system. Therefore, the execution time t_{flush} can be modeled by the following simple equation:

$$t_{\text{flush}} = c_{\text{flush}} \qquad \text{(IV-C.1)}$$

where c_{flush} is a system-specific constant execution time which has to be defined through calibration. To this end, the Predictor initially executes a calibration function on the otherwise idle system calling glFlush in a loop, measures the execution times, and calculates c_{flush} as the average execution time.

D. Prediction Model: CLEAR Command

Next, we consider a more complex prediction model of a context-sensitive command, namely, the CLEAR command. The CLEAR command sets the active render buffer to the color previously specified by the glClearColor command. Moreover, the CLEAR command takes as parameter a bit mask that specifies, which one of the three possible buffers color buffer, depth buffer, or stencil buffer should be cleared. We assume that the execution time to clear a buffer linearly depends on the render buffer size in pixels denoted as s_{rb}. Moreover, the number of bits per pixel influences the amount of data that has to be transferred to memory per pixel. Since we do not know this pixel depth for the three buffer types, nor can we ignore possible side-effects, we calibrate the time to write one pixel for each possible value of the bitmask (*btypes*). Each of these calibrated values is constant for a given system. Namely, we calibrate c_{color}, c_{depth}, c_{stencil}, $c_{\text{color\&depth}}$, $c_{\text{color\&stencil}}$, $c_{\text{depth\&stencil}}$, and $c_{\text{color\&depth\&stencil}}$. Then, the predicted time to clear a certain set of buffers can be modeled as

$$t_{\text{clear}}(btypes, s_{\text{rb}}) = c_{\text{btypes}} * s_{\text{rb}} \qquad \text{(IV-D.1)}$$

If no bit is set in the mask *btypes*, no buffer is cleared, and we assume the execution time to be zero. Otherwise, we calculate the clear time for the given set of buffers *btypes* using the size of the currently active render buffer, according to Equation IV-D.1.

E. Prediction Model: DRAW Command

The most challenging command in terms of execution time prediction is the DRAW command since it depends on various context parameters and has a complex multi-step processing model. The processing of a DRAW command follows a pipeline model as depicted in Fig. 3. The execution time

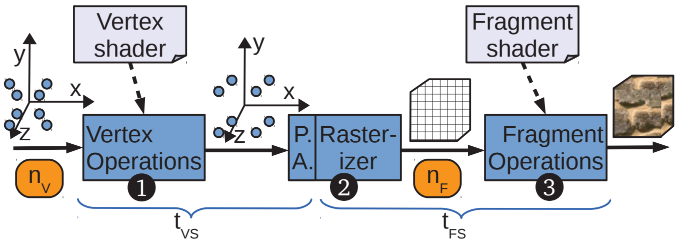

Figure 3. OpenGL draw – main pipeline steps

heavily depends on the shader programs used. The application uses the OpenGL API to create so-called *programs*, where each program is basically one vertex shader and one fragment shader linked together. Before drawing, an application activates one of the created programs with glUseProgram. The draw pipeline takes as input a set of n_v vertices, which can be either provided as vertex buffer object (VBO) or as a vertex array. At the first stage (❶ in Fig. 3), the vertex shader of the active program is then executed for each of these vertices. The vertex shader transforms the vertex position, typically, using a 4×4 model view projection matrix, which is multiplied with each vertex to move, resize, and/or rotate the vertex. In the second stage (❷), the vertices are processed by the Primitive Assembly (P. A.), which prepares for rasterization, e.g., by using face culling, the facing of triangles can be used to skip them if they do not influence the rendered image.

Then, the Rasterizer calculates the pixels (i.e., fragments) of the renderbuffer that are covered by primitives (e.g., triangles). The number of fragments created by the Rasterizer is denoted as n_F. For each of these fragments, the active fragment shader program is executed in the third stage (❸) to assign colors to pixels, for instance, using textures. Additionally, the Fragment Operations step applies a couple of post-processing steps like the depth test or blending to the fragments. Finally, the output is used to update the render buffer. In order to predict the execution time of the DRAW command, we use the following model:

$$t_{\text{draw}}(n_V, n_F, VS, FS) = t_{\text{VS}}(n_V) + t_{\text{FS}}(n_F) \qquad \text{(IV-E.1)}$$

As can be seen from equation IV-E.1, we basically sum up the execution times of the Vertex Operations using the given vertex shader (VS) and of the Fragment Operations using the given fragment shader (FS), which linearly depend on the number of vertices n_V and fragments n_F, respectively. As a simplification, we assume that the execution time of the Primitive Assembly linearly depends on n_V and therefore is considered as part of t_{VS}. Additionally, we assume that rasterization and fragment post-processing depend linearly on t_{FS}. Therefore, our model does not consider them as explicit terms but as part of the execution times t_{VS} and t_{FS}. These assumptions provide a reasonable abstraction of the execution time. Typically, OpenGL ES applications implement their main rendering functionality using sophisticated shaders. Therefore, the impact of the post-processing steps is usually comparatively small, cf. Sec. VI.

In order to evaluate the execution time model of the DRAW command, we need definitions of the vertex processing and fragment processing functions $t_{\text{VS}}(n_V)$ and $t_{\text{FS}}(n_F)$ as well as the input parameters n_V and n_F. If an OpenGL program is used for the first time (or relevant rendering pipeline parameters changed), we initially calibrate t_{VS} and t_{FS}. The function that performs the calibration uses a dedicated OpenGL context that is used for calibration only. The function then compiles copies of the native fragment and vertex shaders, and determines shader attributes and uniforms using the respective OpenGL commands. OpenGL ES 2.0 does not allow to selectively disable fragment processing or vertex processing. Therefore, the function needs to calibrate using all rendering pipeline steps. We use two calibration setups: the first where the Fragment Operations are as minimal as possible to calibrate the Vertex Operations, and the second where the Vertex Operations are as minimal as possible to calibrate the Fragment Operations.

To calibrate t_{VS} the function creates a VBO that contains triangles which—when rasterized—do not result in any fragment. We use glDrawArrays in a loop and then calculate the average execution time caused by one vertex.

To calibrate t_{FS} we create another VBO which consists of triangles of which each covers half of the render buffer used for calibration. Again, we use glDrawArrays in a loop to estimate the average execution time caused by one fragment. The evaluation of the fragment shader function $t_{\text{FS}}(n_F)$ requires knowledge of the number of input fragments n_F, which are dependent on the output of rasterization. Therefore, in contrast to n_V, n_F is not available a priori when the DRAW function is called. Theoretically, we could emulate the vertex shader and rasterization on the CPU to get an exact value of n_F. However, this introduces high CPU overhead. Therefore, we use a less compute-intensive heuristic to estimate n_F. The basic idea is to define a bounding box including all vertices (cf. Fig. 4), and perform the geometric transformation of the vertex shader

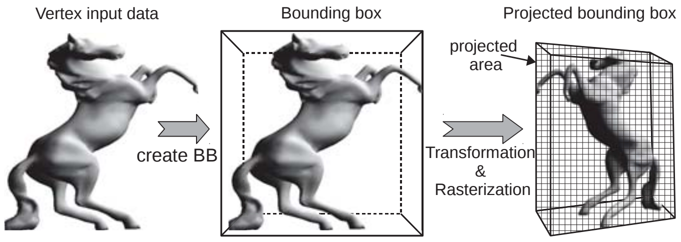

Figure 4. Bounding box applied on a horse model

only on the eight vertices of the bounding box. Then, we calculate the number of pixels that cover the rasterized 2D area (projection) of the 3D bounding box. This number gives us the estimation of the upper bound $\hat{n}_F$ for n_F since the pixels of the actual 3D object might not cover the complete bounding box (cf. Fig. 4). Therefore, depending on the 3D model and transformation, $t_{\text{FS}}(\hat{n}_F)$ might overestimate the execution time.

F. Measuring GPU Execution Time

To calibrate our models, we use execution time measurements of GPU commands as described above. To this end, we need to know when the GPU execution of a command starts and when it has finished. Doing these measurements in user-space without real-time priorities would be inaccurate when the CPU is loaded. One option to increase accuracy would be to give the user-space process real-time priority. However, due to context switches, the measurements would be of limited precision. Moreover, accuracy would still be limited if many real-time processes are executed. Therefore we argue that timestamps should be taken as close to the GPU as possible, i.e., in the kernel-space driver.

In order to detect in kernel-space when a GPU command has finished, we use a similar technique as described in [10]: we let the GPU acknowledge the execution of each command group. To this end, we attach a GPU instruction to each command group that makes the GPU create an interrupt each time a command group has finished execution. Therefore, we have to distinguish between two cases. In the first case, at least one other command is in the queue before the command to be measured. In this case, we measure the finishing time of the previous command, which is at the same time the starting time of the command to be measured. In the second case, the command to be measured is submitted to an empty queue and starts execution immediately. In order to cover this case, we additionally take the time, when the command group was submitted to the queue. If this time is later than the finishing time of the previous command, the queue was empty and we take the submission time as starting time; otherwise, we take the finishing time of the previous command as starting time.

V. IMPLEMENTATION

We have implemented our concept on Fedora 17 running Linux kernel version 3.2.17-1.rt28.1. The Linux realtime kernel is optimized for low latency[1]. Since we perform time measurement within an interrupt service routine, kernel latency directly affects precision of timestamps.

The framework extends the existing Linux architecture by three components:

1) LibETP is a shared library that intercepts OpenGL calls and performs the prediction.
2) A patched version of the drm_nouveau shared library which always notifies libETP before a command group is flushed to the kernel.
3) The kernel module drm_etp which measures the execution time of each command group.

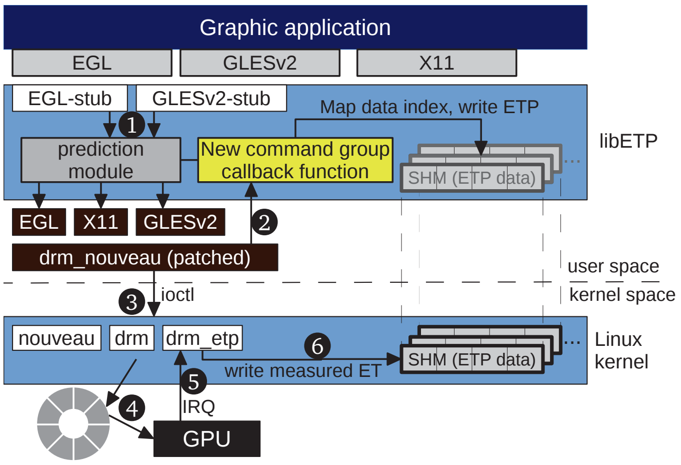

Figure 5. Implemented framework architecture

The architecture of our framework is depicted in Fig. 5. To exchange data between libETP and the kernel module, we use a shared memory segment (ETP data). This shared memory is allocated by the kernel module and can be mapped into user space using a *mmap* call on the modules' character device. The shared memory segment is viewed as an array of structs where each struct contains the predicted execution time (assigned in user-space) and the measured values (assigned in kernel-space). The segment index represents the array index containing the data for the arriving command group. The user-space part of the prediction is encapsulated in a shared library that is binary compatible to the libGLESv2 (for OpenGL ES 2.0) and the libEGL (for EGL) libraries. It intercepts all API calls of these libraries and traces them within the prediction module (❶ in Fig. 5) to keep track of the current OpenGL context and subsequently forwards the calls to the native driver libraries. Eventually, the native driver wants to flush the current command group to the kernel which first invokes the callback function in libETP (❷). The callback function flushes the execution time prediction and writes it to the next unused shared memory segment. After the callback function returns, the system call, which also passes the shared memory segment index number, is performed (❸).

LibETP holds a prediction module which contains estimated values and prediction models for the different OpenGL commands and also the current set of unflushed commands. If the native OpenGL driver decides to flush, the libraries' new command group callback function is called which uses the set of unflushed commands to calculate the estimated execution time and chooses the next free shared memory segment index.

Each time the kernel receives a syscall with a new command group, it forwards that call and the attached shared memory segment index to drm_etp. Before sending a command group to the GPU (which is labeled as ❹), the kernel driver appends a SERIALIZE and a NOTIFY GPU operation—like it was proposed in [10]—which causes the GPU to send an interrupt after that command group has been executed (❺). The interrupt service routine in drm_etp writes the current time stamp to the appropriate shared memory segment field (❻).

Next, we describe more detailed how libETP is implemented, and especially how the prediction module works. When the application issues its first EGL/OpenGL command, libETP performs the following steps:

- Initialize measurement traces and log file output
- Initialize internal data structures and prediction cache
- Map native OpenGL ES 2.0 and EGL symbols
- Map the SHM (ETP data) from the drm_etp kernel module
- Register the callback function at patched libdrm_nouveau

After that libETP traces all relevant EGL and OpenGL calls to internally keep the current state.

Since GPU vendors do not provide detailed information about their hardware architecture, especially an execution time model, the prediction must be based on an execution time model and calibrating its parameters using different dedicated benchmarks. More precisely, to calibrate a parameter libETP creates a sequence of OpenGL commands that is suitable to determine the parameter using the measured execution time. This sequence of commands is passed to the native driver and finally glFlush is called. This results in a single command group that contains just these commands. For the sake of better measurement accuracy, this can be repeated multiple times. After that, libETP sleeps until the kernel has finished all submitted command groups. This is detected by checking if the kernel has assigned the measured execution time to the last recently used shared memory segment. After GPU execution has finished, libETP reads the measured execution time from the respective shared memory segments of the ETP data.

A call for eglInitialize is used to keep the current EGLDisplay handle. The call eglChooseConfig contains the number of bytes per pixel for color, depth, and stencil buffer. When the application calls eglCreateWindowSurface, before returning to the application, libETP creates an internal data structure entry for the native surface. It contains the relevant data, e.g., width and height of the window, and a pointer to the native window. Based on these values it calculates the predicted execution time needed to clear (i.e., assign a specified color to all pixels) that window. This predicted execution time is used for predicting subsequent calls of glClear.

The first time eglCreateWindowSurface is called, additionally the calibration of CLEAR is executed in order to obtain the GPUs memory write speed using glClear. For the calibration a dedicated window surface and OpenGL context are created.

[1]We measured the maximum delay of the kernel using the command "cyclictest -l100000000 -m -Sp99 -i200 -h200 -q 2" while doing our evaluations and measured an average latency of 3 μs and a maximum latency of 40 μs

For the calibration we use a render buffer size of 1200 by 1000 pixels. For this window we call 100 times glClear followed by a glFlush and measure the execution time for the resulting command group. Moreover we measured the execution time for just one glFlush. We executed both measurements (glClear and glFlush) 20 times, and took the averages respectively.

We have implemented the calibration and bounding box models presented in Sec. IV. For the calibration we use the same render buffer size and context as for calibrating glClear. For the calibration of the execution time of Vertex Operations t_{VS} we draw 1000000 vertices that are rasterized to 0 fragments. For calibrating the execution time of Fragment Operations t_{FS} we draw 100 triangles, each covering half of the render buffer, we therefore render 100×600000 fragments. Almost all vertex shaders for OpenGL ES 2.0 multiply the vertex position by a model view projection matrix, e.g., by using $gl\_Position = ModelViewProjectionMatrix * vec4(position, 1.0)$;. This effectively makes emulating the vertex shader a comparatively simple task, since just a single matrix multiplication has to be performed on the CPU. The coverage factor which specifies the expected ratio between the area covered by the projected bounding box and the number of fragments processed during rendering on the GPU significantly influences the measurement accuracy. For many scenarios, a coverage factor of about $50\,\%$ seems to be adequate. For instance, in the example in Fig. 4 about $30\,\%$ of the projected bounding box area is covered by a rendered pixel. However, part of pixels were rendered by multiple fragments, namely, areas where one part of the horse overlaps another like the legs or the head areas. In our current evaluations we use a constant coverage factor of 0.5. We are aware, that this model might not be accurate enough for all scenarios or prediction accuracy requirements. Good mechanisms to obtain the coverage factor need to be developed as part of future work.

VI. Evaluation

In this section, we present our evaluation setup and the evaluation results for each of the mentioned GPU commands.

A. Setup

For evaluation we used an Intel Core i7-3770K CPU and a nVidia Quadro 400 graphics adapter. As mentioned earlier, the GPU commands relevant for the rendering loop are FLUSH, CLEAR, and DRAW. In the implementation of our framework the execution time prediction is performed by libETP.

To allow for an accuracy comparison with libETP, we additionally implemented the *history-based prediction* suggested in [10]. More precisely, after a command group was executed, we insert the binary command group data and its measured execution time into a hash table. For prediction, we look up the command group content in the hash table. If it is available, the stored execution time is used as prediction. Otherwise, the maximum execution time that occurred so far, is used as prediction. Initially, the maximum is set to 0 s. At preliminary evaluations, we observed that each FLUSH command sequence contains a sequence number. This sequence number makes each command group consequently unique. The history-based prediction would rarely find the same command group content in the hash table. Therefore, a naïve history-based prediction would almost always predict the worst-case execution time. We therefore decided to make the history-based prediction ignore sequence numbers of FLUSH commands for a fair comparison.

B. Measuring Accuracy

In this section, we evaluate the accuracy of execution time measurements using the proposed concept, since precisely measuring execution time is key in order to achieve good prediction results. To determine jitter of the measured execution time, the evaluated command groups should be very fast and of low complexity. Therefore, we evaluated the execution time of command groups that contain nothing besides a single FLUSH command, and, inherently, the SERIALIZE and a NOTIFY commands explained in Sec. V. To this end, we created a test program that calls glFlush() 1000000 times and measured the execution time on the GPU. To analyze how these measured values deviate from each other, we depict in Fig. 6 their frequency distribution. From these results, we observe that the

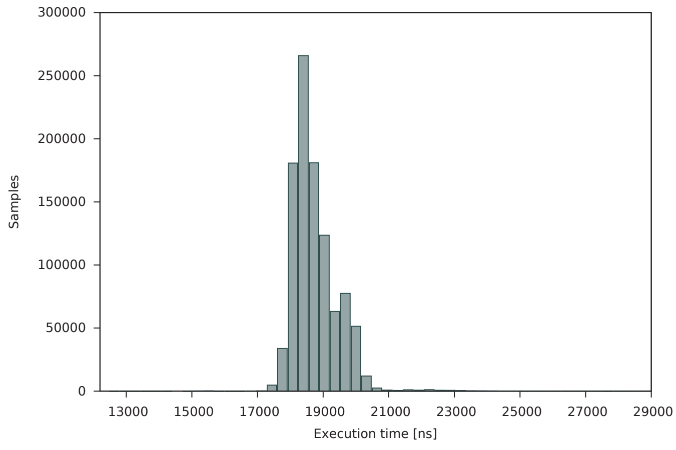

Figure 6. Flush distribution

peak was at about $18.8\,\mu$s with a standard deviation of $864\,$ns and more than $95\,\%$ of the values are in the interval between $18\,\mu$s and $20\,\mu$s. The reason for the observed fluctuations is a combination of jitter in GPU hardware, delayed interrupt processing in the Linux kernel, and DMA delays. However, the precision of the measurement is good enough for GPU scheduling, assuming tens of applications displayed at $60\,$Hz screen refresh rate.

C. Prediction Accuracy: CLEAR Command

The CLEAR command is used in almost all OpenGL rendering programs and typically consumes a significant amount of GPU execution time. In this section, we analyze the prediction accuracy of the CLEAR command. As mentioned earlier, the CLEAR command writes a constant color value to all pixels of a set of buffers. Its execution time therefore depends on the amount of written data and the GPU memory write speed. Analyzing the prediction accuracy therefore mainly answers the question whether the GPU memory write speed is stable enough in order to precisely predict the execution time of CLEAR. For our evaluations we used the Clearspd test program from Mesa [18]. Clearspd uses a large number of CLEAR and FLUSH commands, making it a good choice to measure the prediction accuracy of CLEAR. For compatibility reasons, we changed the initialization code of Clearspd to the OpenGL ES 2.0 API. In Fig. 7 we depict the cumulative distribution function of the measured execution time for the first 10000 command groups. For libETP, the deviation was within the

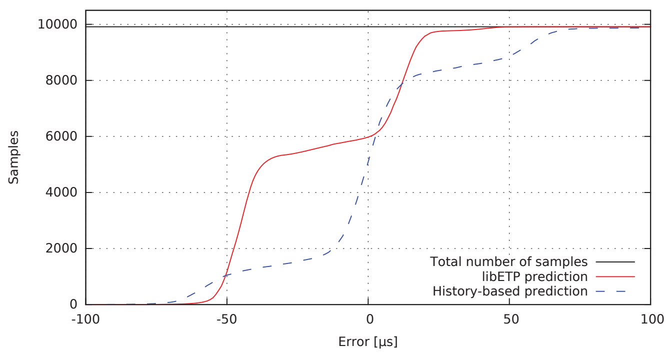

Figure 7. Clearspd CDF

interval $-80\,\mu$s and $+55\,\mu$s. This is small, since on average a command group took $23.4\,$ms to execute. However, at the history-based prediction, 64 command groups were not found and the worst case execution time was assumed for them. This basically means that for all of the 10000 command groups in our evaluations, only 64 different command groups occurred. This is due to the fact that Clearspd initializes the clear color always with black and then iterates it sequentially. Therefore, many command groups are binary equivalent to some other command group that already was submitted. Thus, the original Clearspd program represents a fully deterministic program with lots of repetitions.

Therefore, to better reflect the non-deterministic behavior of GPU accelerated programs, we also evaluated a modified version of Clearspd where the number of glClear loops (i.e., the number of consecutive glClear() calls before glFinish() is called) was chosen randomly between 1 and 500. From the results depicted in Fig. 8, we observe that the number of command groups which could not be predicted correctly by the history-based prediction has increased to 2538. This was due to the fact, that the diversity of the glClear loops significantly increased compared to the original version of Clearspd. Choosing the color randomly, too, would it make almost impossible for the history-based prediction to predict anything besides the overall maximum measured execution time. In a realistic deadline scheduling scenario, drastically overestimating GPU execution time (which happens with the history-based prediction if the worst-case GPU execution time is used) obviously cannot impact other applications with higher priority. However, the overestimated GPU command might effectively stop GPU execution for the respective application if the estimated execution time is too long to fit into time slots available for scheduling. From our results we conclude that the GPU memory write speed is stable enough for precisely predicting the execution time of CLEAR and libETP fits the desired prediction requirements for scheduling.

D. Prediction Accuracy: DRAW Command

In this section, we analyze the prediction accuracy of the DRAW command. We used the bounding box heuristic de-

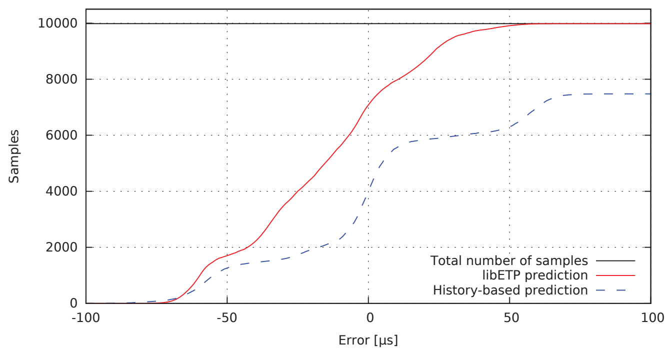

Figure 8. Clearspd Random CDF

scribed earlier and evaluated two popular Linux OpenGL ES 2.0 demo applications. The benchmark application *es2gears* is part of Mesa Demos [18] and renders 3 rotating gears of different color and size. The OpenGL ES 2.0 benchmark application *glmark2-es2* [19] contains multiple scenes; we used the scenes "build" and "shading". For each of these three evaluations we evaluated the absolute prediction error using the bounding box heuristic to estimate the number of fragments n_F. A wrong estimation of n_F could significantly impact prediction accuracy. Therefore, it is important to evaluate if the heuristic is effective for realistic scenarios.

The cumulative distribution function for *es2gears* is depicted in Fig. 9. For each frame, the implementation of

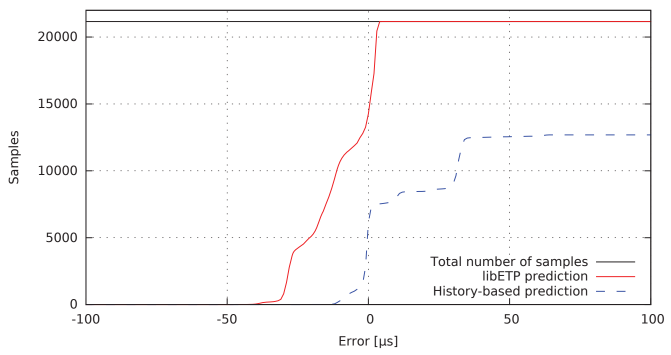

Figure 9. Es2gears distribution

es2gears uses 1 CLEAR command, 280 DRAW commands, and 1 FLUSH command (eglSwapBuffers). Each DRAW command only draws 2 to 5 triangles. However, due to the massive number of calls, the OpenGL driver splits up the commands that render 1 frame into 5 different command groups. The history-based prediction works for command groups three and five. The other command groups contain updates for the two OpenGL Uniform Matrices used by *es2gears*. Since the matrices are not changed in discrete steps, but based on the current time (in order to have a constant rotation speed of the gears), for probabilistic reasons it rarely happens that a rendered frame uses matrices that already occurred in a previous frame (in our evaluations actually never). The libETP prediction deviated no more than $-60\,\mu$s and $+5\,\mu$s.

With *glmark2-es2* we evaluated the benchmark scenes "build" and "shading". The "build" scene renders a rotating horse sculpture, its results are depicted in Fig. 10. The "shading" scene renders a rotating blue cat sculpture, the results are depicted Fig. 11. In contrast to *es2gears*, both scenes of

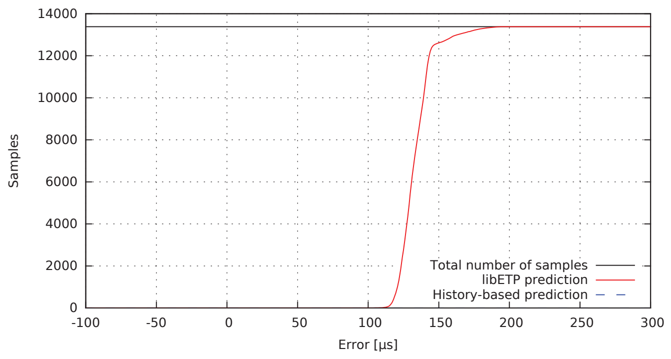

Figure 10. Glmark2-es2 "build" benchmark

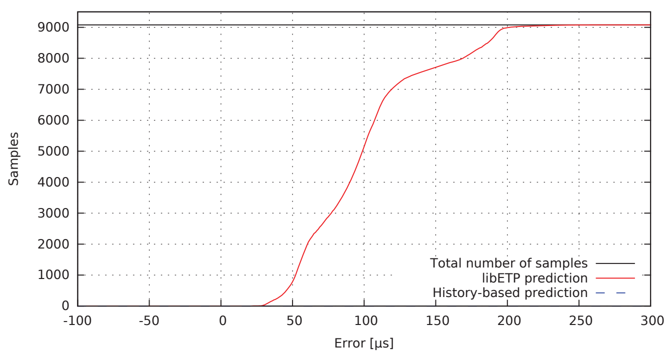

Figure 11. Glmark2-es2 "shading" benchmark distribution

glmark2-es2 use a single DRAW command for each frame, therefore MESA submits all GPU commands of a frame in a single command group. Since all command groups contain the matrices data, which—as with *es2gears*—is different for every frame, the history-based prediction could not predict a reasonable value for any command group. For the "build" scene our results show that the bounding box heuristic of libETP tended to overestimate, on average about $134\,\mu$s. To render a single frame took on average $745\,\mu$s while $879\,\mu$s were predicted by libETP on average. The overestimation of about $18\,\%$ is due to the fact that the actual number of fragments of the rendered horse is less than the assumed $50\,\%$ of the area covered by the bounding box. For the "shading" scene our results show again that the bounding box heuristic tended to overestimate the execution time. Since the scene is more complex, each frame took on average $1099\,\mu$s to render. On average the prediction of libETP was $9\,\%$ above the measured real execution, so the relative error is still comparably low. After OpenGL initialization, the absolute prediction error was within the interval $+20\,\mu$s and $+260\,\mu$s. These results show, that the bounding box heuristic is an effective approach to estimate the number of fragments. Since the number of fragments is one of the key factors for the execution time, it clearly depends on the rendered scene how adequate our heuristic actually is. Moreover, it must be noted, that the current implementation could be deceived by malicious applications that are aware of the used heuristic and submit DRAW commands that are either highly overestimated or underestimated. Further steps are needed to detect such malicious application behavior or at least ensure that the absolute prediction error is limited.

E. *Influence of OpenGL Context*

In Sec. III, we explained our assumption that the OpenGL context affects the execution time of GPU commands. However, this does not necessarily imply that GPU command groups are context sensitive. For instance, if each command group would contain all state-relevant parameters, it could be adequate to ignore the OpenGL context but use the command group data for prediction, as the history-based approach does. In this section we therefore analyze whether using different OpenGL contexts affects the content of the command group data. To this end, we created another version of Clearspd using two contexts. This version of Clearspd creates two window surfaces and OpenGL contexts (instead of one as the original version does). The first window—using the first context—has a resolution of 640x480 and the second window—using the second context—has a resolution of 1920x1080. The main loop uses always black color, 300 loops and alternates between the two contexts. The expected result is that a CLEAR command in the second context takes significantly longer than for the first context, since the render buffer to clear is much bigger. Moreover, just 3 different command group contents occur in this scenario:

A	Enable context 1 (104 bytes) and perform 170 glClear operations (28 bytes each)
B	Enable context 2 (104 bytes) and perform 170 glClear operations (28 bytes each)
C	Perform a postponed FLUSH (20 bytes), perform 130 glClear operations (28 bytes each), finally perform a FLUSH (20 bytes)

The sequence of command groups in this evaluation is A,C,B,C,A,C,B,C, etc[2]. From the results depicted in Fig. 12 we

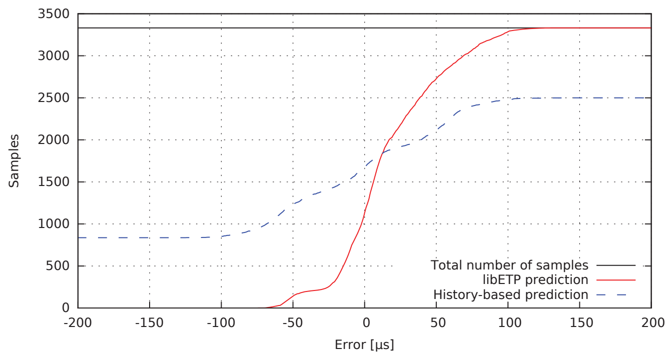

Figure 12. Clearspd Alternating Context CDF

observe, that the history-based prediction predicted completely wrong values for about $50\,\%$ of the command groups. This is due to the fact, that the command group C is assumed to have always the same execution time. However, its execution time actually depends on the active OpenGL ES context, a fact that is completely hidden from the history-based prediction.

[2]In that sequence, as well as in the presented results, we omitted command groups that only consist of a single FLUSH operation for the sake of readability.

Therefore, the assumption that the history-based prediction never clearly underestimates the execution time turned out to be wrong. Furthermore, the assumption that history-based prediction improves its accuracy the longer it runs turned out to be wrong, as well. Since libETP intercepts the eglMakeCurrent calls, it gets aware of context changes and to which window surface each of the glClear calls refer to. Although the execution of a single command group took up to $307\,\mathrm{ms}$ to execute, the prediction error of libETP was never outside the range between $-74\,\mu\mathrm{s}$ and $+130\,\mu\mathrm{s}$. Thus, adding a small safety margin on the predicted value, we could ensure that libETP never underestimates execution time, while still providing a small overestimation. From our evaluations we clearly observe, that using different OpenGL contexts can result in command groups that—although being binary equivalent—have different semantics, i.e., have different execution times caused by the different render buffer sizes. Since libETP is aware of the OpenGL context, it supports execution time prediction also for scenarios that use multiple OpenGL contexts.

VII. Summary and Future Work

In this paper we have motivated that realtime GPU scheduling is necessary to run automotive 3D applications of different safety-criticality on a single platform using a shared GPU. Due to the non-preemptive nature of current GPUs, execution time prediction of GPU rendering commands is required in advance to the scheduling decisions. Our presented framework measures and predicts the execution time of GPU command groups using the OpenGL ES 2.0 API. We presented prediction models for the main GPU commands relevant for 3D rendering, namely, Flush, Clear, and Draw. Our results show that the execution time for Flush and Clear commands can be predicted with high precision. The Draw command is inherently more challenging to predict, since the number of processed fragments is unknown in advance. We therefore use a 3D bounding box of the rendered model and apply the vertex shader projection to heuristically estimate the number of fragments rendered. In our evaluations the absolute prediction error of the Draw command did not exceed $260\,\mu\mathrm{s}$, which shows that our model is an effective way to estimate the number of fragments. In contrast to existing approaches, our framework is fully aware of the OpenGL context. Furthermore, our evaluations show that using a history-based approach on GPU command group level is insufficient for deadline-based GPU rendering.

As part of future work we will analyze how the bounding box heuristic can be improved, especially how the coverage factor can be reasonably used to improve the prediction accuracy. Moreover, we will in-depth investigate the fragment post-processing steps in order to improve the accuracy of our model. Moreover, we will increase prediction accuracy by using live adaptation, i.e., use live measurements to improve slightly wrong calibration measurements later. Furthermore, we will implement a GPU scheduler that uses the predicted values for realtime scheduling of GPU rendering commands.

Acknowledgment

We thank Robert Krutsch from Freescale Semiconductor for his support. This paper has been supported in part by the Automotive, Railway and Avionics Multicore Systems (ARAMiS) project under the research grant of the German Federal Ministry of Research and Education (BMBF), project number 01IS11035R.

References

[1] C. Ebert and C. Jones, “Embedded software: Facts, figures, and future,” *Computer*, vol. 42, no. 4, pp. 42–52, April 2009.

[2] “Nvidia automotive driving innovation,” 2013. [Online]. Available: http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf

[3] (2014) Audi R8 Concept Car. [Online]. Available: http://www.audi.com/content/com/brand/en/vorsprung_durch_technik/content/2014/01/showcar-ces.html

[4] Ford, “Software development kit (SDK),” 2013. [Online]. Available: https://developer.ford.com/develop/openxc/

[5] “New Version of QNX CAR Platform...” 2013. [Online]. Available: http://www.qnx.com/news/pr_5602_1.html

[6] “Mercedes-Benz integration of iPhone App in A-Class,” 2013. [Online]. Available: http://www.iphone-ticker.de/mercedes-benz-iphone-integration-a-klasse-30952/

[7] ISO 26262, *Road vehicles – Functional Safety*. ISO, Geneva, Switzerland, Nov. 2011.

[8] S. Gansel *et al.*, “Towards Virtualization Concepts for Novel Automotive HMI Systems,” in *Proceedings of IESS*, ser. IFIP LNCS. Springer Berlin Heidelberg, 2013.

[9] M. Bautin, A. Dwarakinath, and T.-c. Chiueh, “Graphic engine resource management,” in *Proc. SPIE*, 2008.

[10] S. Kato *et al.*, “TimeGraph: GPU scheduling for real-time multi-tasking environments,” in *Proc. of USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011.

[11] “OpenGL ES; The Standard for Embedded Accelerated 3D Graphics.” [Online]. Available: http://www.khronos.org/opengles/

[12] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

[13] K. Roussos, N. Bitar, and R. English, “Deterministic batch scheduling without static partitioning,” in *Proc. of the Job Scheduling Strategies for Parallel Processing*, ser. IPPS/SPDP ’99/JSSPP ’99. London, UK: Springer-Verlag, 1999, pp. 220–235.

[14] M. B. Jones, D. Roşu, and M.-C. Roşu, “CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities,” in *Proc. of the 16th ACM Symp. on Operating Systems Principles*, ser. SOSP, New York, USA, 1997.

[15] Microsoft WDDM GPU preemption. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/jj553428.aspx

[16] M. Yu *et al.*, “VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming,” in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC ’13. New York, NY, USA: ACM, 2013, pp. 203–214.

[17] Nouveau project. [Online]. Available: http://nouveau.freedesktop.org/wiki/

[18] The Mesa 3D Graphics Library. [Online]. Available: http://www.mesa3d.org

[19] glmark2-es2: OpenGL ES 2.0 benchmark. [Online]. Available: https://launchpad.net/glmark2