Real-time Scheduling for 3D Rendering on Automotive Embedded Systems

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Stephan Schnitzer

aus Stuttgart

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr.-Ing. habil. Roman Obermaisser
Tag der mündlichen Prüfung:	*27.02.2019*

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2019

Contents

List of Figures

List of Tables

Listings

Abstract

3D graphical functions in cars enjoy growing popularity. For instance, analog instruments of the instrument cluster are replaced by digital 3D displays as shown by Mercedes-Benz in the F125 prototype car. The trend to use 3D applications expands into two directions: towards more safety-relevant applications such as the speedometer and towards third-party applications, e.g., from an app store. Traditionally, to ensure isolation, new automotive functions are often implemented by adding further electronic control units (ECUs). However, in order to save cost, energy, and installation space, all 3D applications should share a single hardware platform and thus a single GPU. GPU sharing brings up the problem of providing real-time guarantees for rendering content of time-sensitive applications like the speedometer. This requires effective real-time GPU scheduling concepts to ensure safety and isolation for 3D rendering. Since current GPUs are not preemptive, a deadline-based scheduler must know the GPU execution time of GPU commands in advance. Unfortunately, existing scheduling concepts lack support for dynamic tasks, periodic real-time deadlines, or non-preemptive execution.

In this work, we present the requirements that apply to automotive HMI rendering. Based on these requirements, we propose a Virtualized Automotive Graphics System (VAGS), which uses a hypervisor providing isolation between different VMs, in particular for the head unit and for the instrument cluster.

Additionally, we present a novel framework to measure and predict the execution time of GPU commands using OpenGL ES 2.0. We propose prediction models for the GPU commands relevant for 3D rendering such as Draw and SwapBuffers. For Draw we present two heuristics to estimate the number of fragments, two concepts to estimate the shader execution time, and an optional online adaption mechanism. The number of fragments is estimated either by the bounding box of the rendered model, on which the vertex shader projection is applied, or by a subset of the triangles that is used to estimate the average size of a triangle. To estimate the shader execution time, we either execute them in a profiling environment with a dedicated

OpenGL ES 2.0 Context, or we use a MARS (multivariate adaptive regression splines) model trained offline. The implementation and evaluation of our framework demonstrates its feasibility and shows that good prediction accuracy can be achieved. For instance, when rendering a popular 3D benchmark scene, less than 0.4 % of the samples were underestimated by more than 100 µs and less than 0.2 % of the samples were overestimated more than 100 µs. The overhead introduced by our prediction is negligible on some scenarios and typically below 25 % on the long-run. The application's initial startup is delayed by only about 30 ms of CPU time when using the most efficient concept.

Moreover, we present a real-time 3D GPU scheduling framework that provides strong guarantees for critical applications while still giving as much GPU resources to less important applications as possible, thus ensuring a high GPU utilization. The proposed concepts for execution time prediction are used to make good scheduling decisions and are required since current GPUs are not preemptive. Our implementation is based on an automotive embedded system running Linux and our evaluations show the feasibility and effectiveness of our concepts. The GPU scheduler fulfills given real-time constraints for a dynamic set of applications submitting arbitrary sequences of GPU command batches. It achieves a high GPU utilization of 99 % in a challenging scenario with 17 applications and fulfills 99.9 % of the deadlines of the highest-priority application. Moreover, scheduling is performed highly efficient in real-time with less than 9 µs latency.

Zusammenfassung

Im Automobilbereich erfreut sich der Einsatz von 3D-Grafik zunehmender Beliebtheit. Beispielsweise zeigte Mercedes-Benz im F125 Autoprototypen, wie analoge Zeiger der Kombiinstrumente durch digitale Displays ersetzt werden. Der Trend, 3D-Anwendungen zu nutzen, geht in zwei Richtungen: Zum einen hin zu kritischeren Anwendungen wie der Geschwindigkeitsanzeige, zum anderen hin zu Drittanbieteranwendungen, die beispielsweise über einen Appstore bezogen werden. Um Isolationsanforderungen zu erfüllen, werden traditionell neue Funktionen im Auto häufig mittels neuer Steuergeräte umgesetzt. Um jedoch Kosten, Energieverbrauch und Bauraum im Fahrzeug zu sparen, sollten alle 3D-Anwendungen eine einzige Hardwareplattform und somit auch eine einzige GPU als gemeinsame Ressource nutzen. Für zeitsensitive Anwendungen wie die Geschwindigkeitsanzeige ergibt sich hierbei die Herausforderung, Rendering in Echtzeit zu gewährleisten. Hierfür sind wirksame Konzepte für das Echtzeitscheduling der GPU erforderlich, welche Sicherheit und Isolation beim 3D-Rendering garantieren können. Da aktuelle GPUs nicht unterbrechbar sind, muss ein Deadline-basierter Scheduler die Ausführungszeit der GPU-Befehle im Voraus kennen. Bestehende Schedulingkonzepte unterstützen leider keine dynamischen Tasks, keine periodischen Echtzeitdeadlines, oder setzen unterbrechbare Ausführung voraus.

In dieser Arbeit werden die für HMI-Rendering im Automobilbereich relevanten Anforderungen beschrieben. Basierend auf diesen Anforderungen wird das Konzept des virtualisierten automobilen Grafiksystems (VAGS) vorgestellt, welches einen Hypervisor nutzt um die Isolation zwischen verschiedenen VMs, insbesondere für die Headunit und die Kombiinstrumente, sicherzustellen.

Des Weiteren wird ein neuartiges Framework vorgestellt, welches die Ausführungszeit von GPU-Befehlen misst und basierend auf OpenGL ES 2.0 vorhersagt. Hierbei werden für die relevanten GPU-Befehle wie Draw und SwapBuffers Vorhersagemodelle vorgestellt. Für Draw-Befehle werden zwei Heuristiken vorgeschlagen, welche die Anzahl der Fragmente abschätzen, zwei

Konzepte, welche die Ausführungszeit der Grafikshader vorhersagen, sowie ein optionaler Echtzeit-Korrekturmechanismus. Die Anzahl der Fragmente wird entweder mittels einer Bounding-Box des gerenderten Modells, auf welche die Projektion des Vertexshaders angewendet wird, abgeschätzt, oder durch eine Teilmenge der gerenderten Dreiecke, welche genutzt wird um die Durchschnittsgröße eines Dreiecks zu ermitteln. Um die Laufzeit eines Shaders abzuschätzen, wird er entweder in einer Kalibrierungsumgebung in einem separaten OpenGL-Kontext ausgeführt, oder es wird ein offline trainiertes MARS-Modell verwendet. Die Implementierung und die Auswertungen des Frameworks zeigen dessen Machbarkeit und dass eine gute Vorhersagegenauigkeit erreicht werden kann. Beim Rendern einer Szene des bekannten Benchmarkprogramms Glmark2 wurden beispielsweise weniger 0,4 % der Messproben um mehr als 100 µs unterschätzt und weniger als 0,2 % der Messproben um mehr als 100 µs überschätzt. Unsere Implementierung verursacht bei langer Ausführung eine zusätzliche CPU-Rechenzeit von üblicherweise weniger als 25 %, bei manchen Szenarien ist diese sogar vernachlässigbar. Der Programmstart verlangsamt sich beim effizientesten Verfahren hierbei lediglich um etwa 30 ms. Auf lange Sicht liegt er typischerweise unter 25 % und ist für manche Szenarien sogar vernachlässigbar.

Darüber hinaus wird ein echtzeitfähiges 3D-GPU-Schedulingframework vorgestellt, welches kritischen Anwendungen Garantien gibt und trotzdem die verbleibenden GPU-Ressourcen den weniger kritischen Anwendungen zur Verfügung stellt, wodurch eine hohe GPU-Auslastung erreicht wird.

Da aktuelle GPUs nicht unterbrechbar sind, werden die vorgestellten Konzepte zur Vorhersage der Ausführungszeit verwendet um prioritätsbasiert Scheduling-Entscheidungen zu treffen. Die Implementierung basiert auf einem automobilkonformen eingebetteten System, auf welchem Linux ausgeführt wird. Die darauf ausgeführten Auswertungen zeigen die Machbarkeit und Wirksamkeit der vorgestellten Konzepte. Der GPU-Scheduler erfüllt die jeweiligen Echtzeitvorgaben für eine variable Anzahl von Anwendungen, welche unterschiedliche GPU-Befehlsfolgen erzeugen. Hierbei wird bei einem anspruchsvollen Szenario mit 17 Anwendungen eine hohe GPU-Auslastung von 99 % erzielt und 99,9 % der Deadlines der höchstprioren Anwendung erfüllt. Des Weiteren wird das Scheduling in Echtzeit mit weniger als 9 µs Latenz effizient ausgeführt.

Acknowledgements

I thank especially my advisor Professor Dr. Kurt Rothermel who supported my research and my work in the distributed systems group. I would like to thank him for his continuous guidance, help, and trust. Additionally, my thank goes to Professor Dr. Roman Obermaisser for his support for my research and for reviewing this thesis. My special thank goes to Simon Gansel for our tight collaboration within our complementary research. I appreciate that he spent lots of time reviewing this thesis and was always helpful and constructive. I also thank my colleagues from the distributed systems group who inspired and supported my research. To name but a few, I thank Dr. Frank Dürr for his support and his excellent feedback on our joint publications. I also thank Dr. Boris Koldehofe, Dr. Muhammad Adnan Tariq, Ruben Mayer, and Florian Berg for supporting my research and sharing ideas.

Moreover, I thank the German Federal Ministry for Education and Research (BMBF) who funded part of my research in the scope of the project ARAMiS with funding ID 01IS11035 and allowed me to present my research on international conferences.

I especially thank my wife for her love and her continuous support over the last years.

This work was only possible by the grace of my god and father who listened to the prayers of me and many friends (Bible, Psalm 66 Vers 20).

1. Introduction

1.1. Overview

Innovations in cars are mainly driven by electronics and software today [EJ09, MGR+14]. 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 3D displays, for instance as shown in the Mercedes Benz F125 prototype car [Mer11]. Although, in the beginning, graphical output was mainly 2D content such as movies or 2D maps, the amount of 3D graphics is steadily increasing [Nvi13]. For example, modern navigation systems display 3D city models [AUD15]. Also, the instruments of the vehicle are rendered 3D objects with reflections and shadows to imitate physical instruments as close as possible. Additionally, 3D rendering allows completely different forms of presentation such as the speed indicator at the F125 prototype car shown on its 3D display [Mer11]. Again, a "bird's eye view" with a virtual 3D model of the car and its surroundings supports the driver during parking [bmw15]. To render such complex scenes with high *frame rate*s, graphics processing units (GPUs) are integrated into cars.

Using 3D rendering in automotive scenarios the GPU is typically accessed concurrently by multiple applications, which is quite different to its use in consumer products where often a single application is rendered in full screen mode. In an automotive environment, typically multiple 3D applications run in parallel and are constrained by ISO standards, automotive design guidelines, legal requirements, and demands specific to the original equipment manufacturer (OEM). 3D applications can be sorted depending on the safety-criticality and importance of their rendering. A few examples for 3D applications are listed next, sorted from high importance to low importance.

- Safety-relevant IC applications such as parking assistant or displaying instruments [Mer11, Nvi13] – stutter-free, latency-bound, high *frame rate*s.
- OEM applications like navigation system – decent quality is important, but low latency and high *frame rate*s are less relevant.
- Third-party software such as a web browser executing WebGL or applications from an app store [For13, QNX13, Dai13] that are not quality-assured by the OEM – best effort, using remaining GPU resources.

In order to execute 3D applications with different requirements, the latest high-end cars use multiple hardware platforms to ensure physically isolated 3D rendering. Additionally, in order to save cost, many features like custom 3D games are not yet available, since they would require additional hardware platforms. Next, we describe these two state-of-the-art methods in more detail.

Figure 1.1.: Audi virtual cockpit screenshot[1]

1.1.1. Multiple hardware platforms

Since some applications for the IC are typically certified with ASIL B[2] while the HU applications are QM[2], it is common practice to physically isolate the IC from the HU platform. Since a few years ago, high-end HU systems are using 3D rendering, e.g., for 3D navigation or 3D menus. A relatively new trend is 3D rendering used for the IC, e.g., by the latest Audi TT car [AUD15, AUD14]; a screenshot of the so-called "Audi virtual cockpit" is depicted in Fig. 1.1.

[1]Source: `http://www.audi.de/content/dam/nemo/models/tt/tt-coupe/my-2017/1300x551-layer-header/1300x551_0005_ATT_D_151004_1.jpg`

[2]The automotive standards [ISO11, ISO 26262] address functional safety for vehicles, which includes risk classification ranging from ASIL D (highest risk) to QM (no safety relevance)

Figure 1.2.: BMW 7 series self-parking surround-view[3]

Moreover, self-parking systems with sophisticated 3D-rendered surround-view, such as in the latest 7 series of BMW [bmw15] (cf., Fig. 1.2), are also implemented using a physically separated hardware platform like [Fre14]. Thus, in today's high-end cars three hardware platforms (for HU, IC, and parking) are integrated, which are potentially using rendering on dedicated 3D GPUs. Next, we describe the second state-of-the-art method used to fulfill the automotive requirements.

1.1.2. Limitations on features

When designing a system, the fact that multiple hardware platforms are used limits the flexibility. For instance, to display HU content such as navigation instructions on the IC display current solutions use a LVDS channel of fixed resolution, which can be displayed at a fixed position on the IC display, only. Thus, getting more flexibility implies increased effort and hardware cost.

Furthermore, executing custom 3D applications from a user-selected app store would require either proper isolation from the rendering of the OEMs' applications on the HU, or yet another hardware platform. To this end, OEMs do not support this, yet. Additionally, rear-seat entertainment displays are typically showing video streams transmitted by the HU. Therefore, they do not allow the rear-seat passengers to run their own 3D applications, since the HU cannot prevent impact on the applications displayed on the main HU display and physically separated 3D-enabled platforms for each rear-seat display seem to be too expensive.

[3]Source: `http://www.bmw.com/_common/shared/newvehicles/7series/sedan/2015/showroom/driver_assistance/7-series-sedan-surround-view-01-en.jpg`

1.2. Goals and Problem Statements

Unfortunately, separate hardware platforms 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. Additionally, hardware consolidation provides unprecedented flexibility on the visibility of the applications graphical output, e.g., animations moving windows between IC display and HU display. Moreover, a consolidated hardware with a shared GPU enables support for uncertified 3rd-party applications installed by the user, thus increasing the number of available applications by orders of magnitudes.

1.2.1. Goals

For future cars, a single hardware platform with a powerful 3D GPU shall be able to render the 3D content of different applications with quite different requirements and different importance. A key requirement for safe GPU sharing in automotive scenarios is to provide real-time guarantees for 3D rendering of safety-relevant applications. For instance, deterministic time bounds for presenting warning messages must be guaranteed and less important applications must not interfere with important applications. More precisely, the following goals must be fulfilled in order to run mixed-criticality 3D applications on a single shared GPU.

Concurrency: Typically, many 3D applications are running in parallel.

Flexibility: The set of running 3D applications is dynamic, applications can join or leave during run-time.

Prioritization: Concerning criticality of 3D rendering, some applications are more important than others.

Desired frame rates: Each application has specific requirements for a (uniformly distributed) *frame rate*. For instance, if an application needs to be rendered with 30 frames per second (FPS), a higher *frame rate* would waste valuable GPU time.

Isolation: 3D rendering of important applications must be guaranteed and not affected by less important applications.

1.2.2. Boundary conditions

Historically, the typical use case for 3D GPUs are running a single trusted 3D application (e.g., a game). While technically, also a CPU could be used for 3D rendering (so-called "software rendering"), a 3D GPU performs this task orders of magnitudes faster by using a highly optimized hardware architecture with many parallel computation units. The GPU renders 3D using a rendering pipeline where first the 3D vertex coordinates are calculated by a vertex shader and then the color of each pixel is calculated by a fragment shader (cf. Sec. 3.1.2). The parameters of the rendering pipeline and the shader programs are provided by the 3D applications.

The recent trend to use 3D rendering also in a web browser via WebGL [Khrc] brought uncertified 3D applications new attention, since allowing uncertified applications to use the 3D GPU for rendering can result in unresponsive graphics. Since current GPUs do not support sufficient preemption—i.e., no upper bound for context switch latency is guaranteed—Khronos [Khrb] (the organization publishing the OpenGL standards) states:

> "If a particular draw call takes a long time to execute, because it contains very many triangles, because the associated shaders are computationally expensive, or for any other reason, the user's system may become unresponsive. This is a longstanding problem in the 3D graphics domain, and is one which has received renewed attention since WebGL has been released, because WebGL allows unknown and untrusted code to access the graphics processor." [Khrd]

In such a case, the suggested solution is to reset the GPU:

> "Solutions already exist to this problem on some operating systems. For example, Microsoft Windows Vista and later support a new driver model which will reset the graphics processor if it spends too long on any particular operation. The WebGL implementation can detect that the graphics card was reset, warn the user that WebGL content might have caused it, and prompt the user if they want to continue running the content." [Khrd]

Unfortunately, for automotive scenarios, resetting the GPU is not an option since it cannot happen without delay and would even require 3D applications to be restarted since their GPU context became inconsistent. To this end, automotive 3D rendering can neither use explicit GPU preemption, nor reset

the GPU to preempt it. For a GPU shader program, this implies that it must always terminate. For OpenGL ES 2.0, the OpenGL ES Shading Language specification [Sim09] in Appendix A.4 forbids *while loops* and allows only *for loops* that can be unrolled at compile-time. The newer OpenGL Shading Language (GLSL) specification for OpenGL ES 3.0 [SKBR12] contains no such restriction. Since the possibility to create non-terminating loops can cause unwanted behavior and system malfunction, forbidding them is common. For instance, the most popular area where untrusted shader code is executed is the WebGL standard [Khrc], which is based on OpenGL ES 2.0. To the extent of our knowledge, all browsers supporting WebGL strictly follow the specification in forbidding loops which cannot be unrolled at compile-time. Since many 3D GPU drivers actually would not reject non-terminating shader source code, the web browser implements a safety layer filtering out potentially non-terminating or extremely long-running code. To this end, for automotive scenarios, only loops that can be unrolled by the shader compiler and thus are guaranteed to terminate can be supported.

1.2.3. Execution time prediction

Without preemption, we explicitly need to consider the execution time of rendering jobs to ensure that low priority (non-safety critical) rendering jobs do not prevent the timely execution of high priority (safety critical) jobs. To this end, a non-preemptive scheduling approach is required. Non-technical approaches to determine the execution time by certification of the 3D 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. Consequently, the execution times of the GPU commands must be predicted prior to their execution.

Existing concepts like [KLRI11] use history-based approaches in kernel space to predict the execution time. While such approaches are easy to implement, they are not aware of the rendering setup and the rendered scene. To this end, they cannot predict the first commands of an application. Additionally, this approach is based on the assumption that the same GPU commands result in the same GPU execution time, which is not always the case since the GPU-internal state depends on the OpenGL context and can differ [SGDR14].

The execution time of a DRAW command depends on many parameters. The most parameters are the used shader programs and the respective number of

instances. For instance, the input parameters of the shader programs can influence the positions of vertices during the vertex shader execution. This changes the number of fragments, which heavily affects the execution time.

The number of vertex shader instances is directly given by the application's 3D API calls. To this end, the main challenges when predicting a DRAW command are to estimate

- the number of fragments generated by the vertex shader and the used attribute data and
- the execution time per shader instance.

Next, we address both challenges in more detail.

Number of fragments. The number of fragments is one of the most relevant factors of accurate prediction, since for each fragment one instance of a fragment shader must be executed on the GPU. Unfortunately, in order to determine the number of fragments accurately before execution on the GPU, the full vertex processing step of the OpenGL rendering pipeline would have to be emulated on the CPU. For medium or large 3D models, this is not feasible without severely affecting rendering performance, since a massive overhead would be introduced into the prediction. Consequently, a heuristic must be used, which inevitably introduces prediction errors (addressed in Sec. 3.5.1).

Execution time per shader instance. An application provides the source code of vertex shader and fragment shader written in the GLSL. The GLSL supports if-statements, for-loops, and while-loops, but no non-structured commands such as "goto". The source code is compiled by the user space GPU driver, which creates a shader binary with the target GPU instruction set. It performs typical compiler optimizations such as factoring out, loop unrolling, dead code elimination, or constant folding. The user space driver is typically proprietary, since it often contains intellectual property of the GPU manufacturer, which means that shader compilation is a black box. To this end, heuristics must be used to predict the execution time per shader instance (addressed in Sec. 3.5.2 and Sec. 3.5.3).

1.2.4. GPU scheduler

The non-preemptive GPU scheduler is responsible to dispatch concurrently running 3D applications such that the goals *prioritization*, *desired frame rate*, and *isolation* are fulfilled. Since the set of applications can change during run-time and the required execution time is determined by execution time prediction during runtime, scheduling algorithms for fixed sets of periodic tasks [Liu69, LL73] are insufficient. The Shortest Process Next (SPN) algorithm (cf. [TB14]) does not support given priorities and *frame rate*s. Existing approaches for 3D GPU scheduling address just fairness [DWA08, BDC08] and optionally weighted fairness with priorities [KLRI11]. However, to ensure a guaranteed latency until a frame is rendered requires a much more sophisticated scheduling algorithm. More precisely, the scheduling algorithm must keep track of the dynamic frame deadlines of each application. Unfortunately, the execution time required for an application to render its frame can change between different frames and is not available to the scheduler before the respective user space process has submitted all of its commands. Furthermore, a reservation policy for future frame periods is needed, since long-running commands of low-priority applications may not only affect the current period of higher-priority applications, but also their future periods. While fulfilling our goals is mandatory, the performance of our scheduling approach is extremely important. The latency experienced by applications and the overhead introduced by the execution of the scheduling algorithm itself thus must be very small. Furthermore, the scheduling decisions shall result in a high GPU utilization to ensure the available GPU resources are exploited. Our approach, presented in Chapter 4, addresses all these challenges—as shown by our evaluations in Section 4.5.

1.3. Project ARAMiS

The contributions of this work were supported by the project ARAMiS [ARA16] of the German Federal Ministry for Education and Research (BMBF) with funding ID 01IS11035. In this section, we provide a brief overview of the ARAMiS project and how its goals are relalated to the contributions of this work.

ARAMiS is short for "Automotive, Railway and Avionics Multicore Systems". Its goal was to build a technological platform to further increase safety, efficiency, and comfort by using multicore technology in the automotive, avionics, and railway domains. ARAMiS ran from December 2011 to March 2015 and had a planned budget of 36 million Euro.

1.3.1. Background

Historically, single-core CPUs were prevailing in PCs, servers, and embedded systems. However, the development of new CPU generations by increasing the clock speed turned out to be slow and inefficient. In many scenarios, computation can be performed in parallel, which means that multicore CPUs can often provide a significant performance improvement compared to single-core CPUs. The rise of 3D rendering with its compute-intensive but also highly parallelizable workload brought up 3D GPUs as specialized multicore processing units. When using multicore systems for the automotive domain, many requirements must be fulfilled, such as real-time, availability, functional safety, and efficiency. Existing concepts typically do not fulfill them and are therefore not suitable. This motivates the goal of ARAMiS to find new architectures, methods, and concepts that allow multicore systems to be used for automotive platforms.

1.3.2. Structure

ARAMiS was organized in multiple subprojects, which are depicted in Fig. 1.3. Next, the subprojects are briefly described, focusing on the results relevant for this work.

TP0 provided the coordination and project management of the overall project

TP1 defined scenarios consisting of use cases and requirements, including the requirements for virtualized 3D rendering.

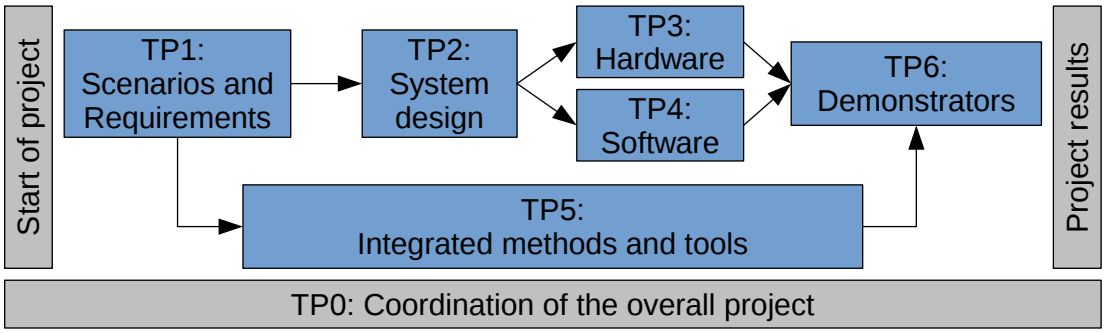


Figure 1.3.: Dependencies of the ARAMiS subprojects

TP2 designed a system for a virtualized 3D rendering that fulfills the requirements of TP1.

TP3 developed hardware concepts, focusing on heterogeneous architectures, security, safety, certifiability, and virtualization. Since the GPU vendors neither grant access to the hardware layout, nor do programmable chips (e.g., FPGAs) provide sufficient performance for 3D rendering, TP3 was not in the focus of this work.

TP4 developed software concepts, which includes virtualization, compositing, 3D execution time prediction, and real-time 3D GPU scheduling, i.e., the main results of this work.

TP5 was a small subproject that examined tools supporting the design of multicore systems. For virtualized 3D graphics, no relevant tools are known.

TP6 built multiple demonstrators (2 for automotive, 2 for avionics, 1 for railway) covering most use cases of TP1.

1.3.3. Results

The results of ARAMiS were published in more than 120 documents, more than 70 scientific publications, and 5 demonstrators. This shows the relevance of multicore systems in the automotive, avionic, and railway domains in general, and the relevance of a virtualized graphics system with real-time 3D scheduling in particular. In particular, we built the automotive cockpit demonstrator VCT-B in collaboration with Daimler. It shows a prototype of virtualized 3D rendering of IC and HU.

1.4. Contributions

In this section, we describe our contributions to the goals presented in Sec. 1.2.

1.4.1. Requirements analysis for graphics virtualization

In Section 2.1, we thoroughly analyze relevant ISO standards and legal requirements and derive seven technical requirements for a virtualized automotive HMI system. Such requirements have been largely neglected by current virtualization efforts, which did not target automotive systems with their specific requirements, in particular, with respect to safety. For OEMs, the certifiability of automotive system functionalities is highly relevant. According to [ISO11, ISO 26262], for each functionality safety-criticality shall be identified and mapped to criticality-classes[4]. To fulfill the criticality-level, the severity and likelihood of failures must be determined using, for instance, *failure mode and effects analysis* (FMEA) [Sta03]. Moreover, certifiability also applies to custom third-party applications. For instance, [ISO02, ISO 15005] prohibits displaying movies to the driver while the vehicle is in motion.

1.4.2. Virtualized automotive graphics system

In Section 2.2, we present a concept for a Virtualized Automotive Graphics System (VAGS). To this end, we elaborate on the challenges that are due to the identified requirements caused by consolidation of mixed-criticality graphics electronic control units (ECUs) as used, in particular, by the HU and IC. Although virtualization is a mature technology for general resources like CPU or main memory, existing concepts do not provide sufficient isolation for accessing shared graphics hardware (GPU). Our proposed architecture uses a dedicated driver-VM, which is used as central instance by the other VMs to present content on the displays. In particular, the driver-VM manages real-time 3D GPU scheduling, display access permissions, and input events.

In Section 2.3, we describe our automotive cockpit demonstrator, which contains the major components of a Virtualized Automotive Graphics System (VAGS). It shows the feasibility of our concepts and how they can be implemented.

[4] [ISO11, ISO 26262] specifies five safety requirement levels: Four ASIL (Automotive Safety Integrity Level) ranging from ASIL-A (low criticality) to ASIL-D (high criticality), and one no-criticality level QM (Quality Management)

1.4.3. Execution time prediction for 3D rendering commands

In Chapter 3, we present a framework for measurement and prediction of the execution time of GPU command batches. Prediction is performed in user space, which gives the huge benefit that context information can be determined much easier, since it can be inferred from the commands transmitted through a standardized API like OpenGL ES. The basic idea is to predict the individual execution time of graphics commands using models that are determined either during runtime or offline.

In particular, we propose models for the main commands relevant for 3D rendering, namely, Flush, Clear, Draw, and SwapBuffers, using the Open Graphics Library for Embedded Systems (OpenGL ES) standard [Khra]. Flush has constant execution time independent of the context. The execution time of Clear (if not integrated into SwapBuffers) 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 estimate the number of fragments and the time the processing time per vertex and per fragment using the given shader program. We achieve this by emulating the vertex shader either on a bounding box of the 3D model, or on a representative subset of triangles. To profile the execution time of these commands on the specific GPU and to execute the emulation, we propose an online approach that instruments the GPU command groups in kernel space on the fly and an offline approach that uses machine learning models based on platform-specific training data. Furthermore, we present a fine-grained online correction to further improve prediction accuracy. We implemented our prediction framework and present evaluation results that compare our approaches with each other and an existing history-based approach. We show that our prediction framework achieves unprecedented accuracy that is sufficient even for challenging scheduling scenarios.

1.4.4. 3D GPU scheduler

In Chapter 4, we present a framework for real-time 3D GPU scheduling. Without preemption, we explicitly consider the execution time of GPU command batches to ensure that low priority (non-safety critical) GPU command batches do not prevent the timely execution of high priority (safety critical) jobs. Our GPU

scheduling algorithm considers in addition to the job execution time several other parameters like the priority of the rendering jobs, screen *refresh rate*, and target *frame rate*. In more detail, we make the following contributions for 3D GPU scheduling:

1. A system architecture and framework for 3D GPU scheduling that uses execution time prediction of GPU rendering jobs.

2. A priority-based real-time scheduling concept that specifically addresses desired *frame rate*s of dynamic rendering jobs and bitblitting aligned to the vertical synchronization of the displays.

3. An implementation of the framework and the proposed 3D GPU scheduling concepts.

4. An evaluation showing the conformance of the implementation compared to the setup, a high GPU utilization of about 97 %, and less than 10 μs scheduling latency.

1.4.5. Further contributions

In Chapter 2.3, we present an automotive cockpit demonstrator (VCT-B) that was developed in collaboration with the Daimler AG in Stuttgart. It uses a consolidated hardware platform for IC and HU, using a hypervisor for isolating the virtual machines that contain HU and IC functionality. The input buttons on the steering wheel and the push-and-rotary switch can be used to navigate through menus and change modes. We demonstrate the uses cases for a VAGS, such as a flexible display usage and isolation. To this demonstrator, the author has contributed concepts, code, and guidance. The concepts include an access control system for display areas, the inter-VM communication layer, efficient compositing concepts, and the virtualization layer.

1.4.6. Related publications and contributors

In this section, we present the scientific publications by the author that are related to this work. For each publication, we briefly describe the amount of the author's contribution. The author was advisor of all diploma, master, bachelor, and study theses cited in this section. We also declare the contributions to this work that are beyond the scope of the referenced scientific publications. All publications have been written in collaboration with the other authors. Especially Simon Gansel

provided lots of valuable feedback to both, the concepts, and the publication texts. The feedback and the discussions with Prof. Dr. Kurt Rothermel and Dr. Frank Dürr helped to tailor and improve the publications.

Focus of this work. In [GSD+13] the requirements for automotive graphics are expounded and the concept of a VAGS is presented. The author's contribution to this publication was 45 %. In [SGDR14] we presented execution time prediction using the bounding box heuristic and profiling of shaders during runtime. The author's contribution to this publication was 85 %, the implementation was sole work of the author. In [SGDR16] we presented the real-time scheduling for 3D GPU rendering. The author's contribution to this publication was 85 %, the GPU scheduler implementation was sole work of the author.

The diploma and master theses of Fabian Römhild, Armin Cont, and Waqas Tanveer [Röm11, Con11, Tan13] gave insight about the scheduling capabilities of OpenGL and CUDA. The observed limitations justified our concept to do GPU scheduling in kernel space. The diploma thesis of Martin Thielefeld [Thi12] improved the knowledge how GPU execution time depends on OpenGL ES 2.0 Context, thus helping to build adequate prediction models. The study thesis of Felix Zehender [Zeh14] helped to better understand how the 3D GPU driver in user space (MESA, in particular) compiles and optimizes shader code. The master thesis of Hua Ma [Ma14] provided a better understanding of the Vivante GPU kernel driver, which helped to implement our Execution Time Monitor. The master thesis of Robin Keller [Kel16] helped to understand the limitations of linear regression regarding GPU execution time prediction. As a consequence, we used a non-linear model without online learning and only for the prediction of shader execution times.

Yaroslav Nalivayko worked as a student assistant on the execution time prediction. He implemented requested features such as saving the prediction parameters to XML, the triangle samples approach, and helped on debugging and creating training data.

Completive to this work. Within the scope of the ARAMiS project, small parts of this work were published in [RAL+15], where virtualization concepts in the scope of ensuring safety and security in automotive systems are described.

The author contributed 10 % to each of the publications about automotive HMI access control concepts [GSGH+14, GSGH+15] and efficient compositing [GSC+15].

Ahmad Gilbeau-Hammoud contributed to [GSGH+14, GSGH+15] with his diploma thesis [GH13] and his subsequent work as a student assistant and research assistant. Riccardo Cecolin contributed to [GSGH+15] with his diploma thesis [Cec14]. The master thesis of Han Zhao [Zha15] proposes a 3D compositor for a VAGS that allows to combine the 3D output of different applications. The depth information is used to determine visibility and shader programs operating on the applications color and depth buffers are used for customized lighting effects. Thus, this thesis provides further motivation for a VAGS on a consolidated hardware architecture. The master thesis of Andrej Eisfeld [Eis14] improved inter-VM communication of the OpenGL ES 2.0 and EGL protocols, showing that efficient transmission of graphics data in a VAGS is possible.

1.5. Structure

The rest of this work is structured as follows. In Chapter 2 the relevant requirements and our architecture are presented. Section 2.1 presents the relevant automotive HMI requirements. The architecture of our proposed Virtualized Automotive Graphics System is described in Section 2.2. To demonstrate the scenarios of our automotive graphics virtualization, we created an automotive cockpit demonstrator, which is described in Section 2.3. Chapter 2 is complemented by related work in Section 2.4 and a summary in Section 2.5.

Our main contributions are the execution time prediction—presented in Chapter 3—and the real-time GPU scheduler—presented in Chapter 4.

Related to execution time prediction (ETP), we provide background information about EGL, OpenGL ES 2.0, and machine learning in Section 3.1. The system model is presented in Section 3.2. In Section 3.3, we describe the prediction architecture and how the prediction models are used.

The rather simple models for Flush, Clear, and SwapBuffers are described in Section 3.4. The challenging Draw command and its sub models to estimate fragments and shaders are presented in Section 3.5. This includes fragment estimation heuristics, performance parameter profiling, and machine-learning-based models. The optional *Online Adaption* allows to correct predictions leaning to either overestimation or underestimation and is presented in Section 3.6.

The implementation is expounded in Section 3.7 and the evaluation results are presented and discussed in Section 3.8. The related work for execution time prediction in Section 3.9 is followed by a summary and an outlook on future work in Section 3.10.

For GPU scheduling we explain the requirements in Section 4.1 and the system model in Section 4.2. The concepts are explained in Section 4.3 and followed by a description of the implementation in Section 4.4. In Section 4.5 we present our evaluation results, which show feasibility, effectiveness, and performance of our GPU scheduler. The chapter is concluded an outlook on preemptive GPU scheduling in Section 4.6, related work in Section 4.7, and the summary and future work in Section 4.8.

This work is concluded in Chapter 5.

2. Requirements and Architecture

In this chapter, we thoroughly analyze relevant ISO standards and legal requirements and derive seven technical requirements for a virtualized automotive HMI system. Such requirements have been largely neglected by current virtualization efforts, which did not target automotive systems with their specific requirements, in particular, with respect to safety. For OEMs, the certifiability of automotive system functionalities is highly relevant. According to [ISO11, ISO 26262], for each functionality safety-criticality shall be identified and mapped to criticality-classes[1]. To fulfill the criticality-level, the severity and likelihood of failures must be determined using, for instance, *failure mode and effects analysis* (FMEA) [Sta03]. Moreover, certifiability also applies to custom third-party applications. For instance, [ISO02, ISO 15005] prohibits displaying movies to the driver while the vehicle is in motion. These specific regulations impose challenging technical requirements to virtualization. To this end, we elaborate on the challenges that are due to the identified requirements to consolidate mixed-criticality graphics ECUs as used, in particular, by the HU and IC. Although virtualization is a mature technology for general resources like CPU or main memory, existing concepts do neither provide sufficient isolation for accessing shared graphics hardware (GPU) and input devices (e.g., steering wheel buttons), nor do they provide sufficient isolation for implementing the flexible presentation of application windows.

This chapter is structured as follows. In Sec. 2.1, the requirements for automotive graphics systems are analyzed and seven technical requirements derived. In Sec. 2.2, we propose the architecture for virtualized automotive graphics. The automotive cockpit demonstrator VCT-B is explained in Sec. 2.3, followed by related work in Sec. 2.4, and a summary of this chapter in Sec. 2.5.

[1] [ISO11, ISO 26262] specifies five safety requirement levels: Four ASIL (Automotive Safety Integrity Level) ranging from ASIL-A (low criticality) to ASIL-D (high criticality), and one no-criticality level QM (Quality Management)

2.1. Requirements

In this section, we discuss requirements that are relevant for automotive HMI systems. Automotive application development is constrained by ISO standards, automotive design guidelines, legal requirements, and OEM specific demands. The design guidelines (e.g., [AAM06, AAM 2006], [ESO08, ESoP 2008], [JAM04, JAMA 2004]) in the automotive domain are almost completely derived from the following ISO standards.

- [ISO96, ISO 11428] Ergonomic requirements for the perception of visual danger signals.
- [ISO02, ISO 15005] Requirements to prevent impairment of the safe and effective operation of the moving vehicle.
- [ISO04, ISO 16951] Priority-based presentation of messages.
- [ISO10, ISO 2575] Symbols for controls and indicators.
- [ISO08, ISO 15408-2] Security in IT systems.
- [ISO11, ISO 26262] Risk-based assessment of potentially hazardous operational situations and of safety measures.

In the following, we propose seven technical requirements for automotive HMI systems. For each of them we added references to relevant sections of the mentioned ISO standards.

2.1.1. R1 – Input Event Handling

R1.1 – Restricted Access Control: For user input events *access control* is required and it shall not violate any of the following constraints [ISO02, ISO 15005]. Applications using dialogues shall not require to use input devices in a way that demands removal of both hands from the steering wheel while driving (5.2.2.2.2). Additionally, exiting a dialog or an application shall always be possible (5.3.3.2.1) unless legally required or traffic-situation-relevant (5.3.3.2.3).

R1.2 – Restricted Processing Time: A *maximum processing time* for input event handling shall be met. For instance, response to tactile user inputs shall not exceed 250 ms (5.2.4.2.3).

2.1.2. R2 – Restricted Window Creation and Positioning

R2.1 – Restricted Visibility of Windows: Usually, graphical applications use API functions to change the *visibility of windows*, e.g., to create, hide, or position them. This functionality must be restricted, and functions not intended to be used by the driver must be inaccessible for him [ISO02, ISO 15005] (5.2.2.2.4).

R2.2 – Priority-based Displaying of Windows: If multiple windows shall be displayed, the importance of each of them must be defined. Importance is represented by priorities, which can depend on safety requirements and software ergonomic aspects (5.2.4.2.4) that must be met by the system (5.2.4.3.3). Moreover, they can depend on urgency and criticality, which have to be defined [ISO04, ISO 16951] (3.5). Additionally, appropriate reactions (e.g., behavior in case of conflicts) shall be enforced [ISO04, ISO 16951] (Annex B). Furthermore, country-specific legal requirements constrain the definition of the priorities, e.g., German law requires the constant visibility of the speedometer while the vehicle is in motion (StVZO §57 [Jan11]). Additionally, visual information must be presented in a consistent way [ISO02, ISO 15005] (5.3.2.2.1).

R2.3 – Timing Constraints: An automotive HMI system shall enable applications to provide important information to the driver within given *time constraints*. This means that windows showing information shall be visible within given time constraints [ISO02, ISO 15005] (5.2.4.3.4). If applications require user interaction, e.g., if a user selects a radio channel, the flow of information must not adversely affect driving (5.2.4.2.1). Concretely, according to [AAM06, AAM 2006] Section 2.1, each glance shall not exceed 2 seconds. Hence, any kind of animation shall not run longer than 2 seconds.

2.1.3. R3 – Trusted Channel

R3.1 – Integrity and Confidentiality: In environments where applications run inside VMs, communication is inevitable. This holds for communication that previously used dedicated communication hardware and is now replaced by software-based inter-VM communication. According to [ISO08, ISO 15408-2], communication between applications and hardware must provide integrity and confidentiality, for both, user data

(14.5.8.2) and software components providing relevant functionality (17.1.5.3). All applications that need trusted communication shall be able to use it (17.1.5.2).

R3.2 – Authentication and Non-Repudiation: Identification shall be assured even between distinct systems (17.1.5.1), which also applies to inter-VM communication. A trusted channel also requires non-repudiation of origin (8.1.1 and 8.1.6.1-3) and receipt (8.2.1 and 8.2.6.1-3). This requires authentication and may also involve cryptographic key management (9.1.1) and key access (9.1.7.1).

2.1.4. R4 – Virtualized Graphics Rendering

In our system, multiple VMs have shared access to a single GPU, and therefore the VMM has to provide isolation. That is, unintended interference between applications must not occur.

R4.1 – Priority Handling: Application windows must be assigned a priority, which determines how GPU commands are processed [ISO02, ISO 15005] (5.2.4.2.4 and 5.2.4.3.3), [ISO08, ISO 15408-2] (15.2.5.1-2 and 15.2.6.1-2). For instance, a rendered speedometer must have a high priority, since the German law regulates that it must be visible while driving and display the current speed (StVZO §57 [Jan11]).

R4.2 – Rendering Time Constraints: Not only comparative requirements (like priorities) but also absolute timing requirements have to be fulfilled. A response to a drivers tactile input shall not exceed 250 ms [ISO02, ISO 15005] (5.2.4.2.3). Similarly, emergency signals may require constant redraw rates to represent flashing lights [ISO96, ISO 11428] (4.2.2). This requires appropriate CPU and GPU resources and imposes a minimum *frame rate* since the delay between two consecutive frames is constraint by an upper bound. The upper bound must be known to determine the effectiveness of safety-critical messages [ISO04, ISO 16951] (Annex F) and also to allow for the definition of delays after which messages are displayed (Annex B). Additionally, OEMs (especially of premium brands) have demanding requirements for the rendering, e.g., that the speedometer shall be rendered stutter-free at 60 frames per second.

R4.3 – GPU Resource Isolation: The GPU is a controlled resource according to [ISO08, ISO 15408-2]. To prevent unintended interference, it must be

possible to provide guarantees to certain applications that they are provided sufficient GPU resources such as processing time. Therefore, it must be possible to control which GPU resources individual windows, graphical applications, or VMs are allowed to use (15.3.6.1 and 15.3.7.1-2).

2.1.5. R5 – Reconfiguration of Policies

A set of permissions that apply to user input events, application windows, and the related scheduling and isolation is called a *policy*. At each point in time, exactly one policy is active, though policies are dynamically switched during runtime depending on the system state.

R5.1 – Dynamic State Changes: In accordance to [ISO02, ISO 15005], a *state change* happens either on user request or automatically by system-defined rules. A state can depend on a current vehicle condition like "vehicle is in motion", which could require the deactivation of applications that are not intended to be used by the driver while the vehicle is in motion (5.2.2.2.4). Otherwise, an automotive HMI system shall provide sufficient information and warnings to provide the driver with the intended purpose in a current state. For every state change, specified *deadlines* apply to determine a consistent and accurate transition between different states. The definition of states and system behavior is explained in more detail in [ISO04, ISO 16951] (3.3 and Annex E).

R5.2 – Dynamic Policy Changes: Authorized software components shall be enabled to apply changes to policies during runtime. This includes granting and revoking permissions on both, currently active and currently inactive policies. As for R5.1, deadlines apply to dynamic policy changes. Where applicable and allowed, the driver shall be able to change the active policy to manipulate the flow of information [ISO02, ISO 15005] (5.3.3.2.3).

R5.3 – Presentation Enforcement: The system-defined rules shall enforce the presentation of legally required messages and traffic-situation-relevant messages. Presentation requires that those messages are visible and perceivable, in particular, if state changes require driver attention [ISO02, ISO 15005] (5.3.2.2.2). Furthermore, state-related information shall be displayed either continuously or upon request by the driver.

2.1.6. R6 – Certifiability

For an OEM, certifiability is an essential part of the software development process, e.g., by using methods like FMEA [Sta03]. The development process for certified software, in particular, for high criticality levels, is quite complex and expensive. A key indicator for complexity is the number of function points that correlates with the approximated number of software defects [EJ09]. Hence, a system shall be developed with respect to an easy certification according to [ISO11, ISO 26262].

2.1.7. R7 – System Monitoring

System Monitoring puts the focus on logging, detecting, and reacting to events that possibly are relevant to provide safety.

R7.1 – Secure Boot: Derived from [ISO08, ISO 15408-2], the system shall provide *secure boot* to ensure the integrity of the system. Compromising the system (14.6.9.1) or system devices or elements (14.6.9.2) by physical tampering shall be unambiguously detected.

R7.2 – Auditing: The *auditing* of all safety-critical related events shall be guaranteed to ensure traceability of system activities in an automotive HMI system that potentially violate safety or security. Therefore, direct hardware access must not be permitted to ensure that auditing cannot be bypassed. For a potential violation analysis, a fixed set of rules shall be defined for a basic threshold detection, [ISO08, ISO 15408-2] (7.3.2). To indicate any potential violation of the system-defined rules, the monitoring of audited events shall also be based on a set of rules (7.3.8.1) that must be enforced by the system either as an accumulation or a combination of a subset of defined auditable events that are known to threat the system security (7.3.8.2). Similarly, all changes to policies initiated by applications shall be monitored and verified.

R7.3 – Supervision of Timing Requirements: It is a requirement to regulate the flow of information to ensure short and concise groups such that the driver can easily perceive the information with minimal distraction [ISO02, ISO 15005] (5.2.4.2.1). Therefore, specified *time restrictions* need to be verified. This also includes the auditing of driver tactile input and system response time, which shall not exceed 250 ms (5.2.4.2.3).

R7.4 – Detection of DoS Attacks: The occurrence of any event representing a significant threat such as a *DoS attack* shall be detectable by the system in real-time or during a post-collection batch-mode analysis [ISO08, ISO 15408-2] (7.3.2).

R7.5 – Perception of Visual Signals: For the perception of visual danger signals, visibility properties like fractions of luminances [ISO96, ISO 11428] (4.2.1.2) and colors of signal lights (4.3.2) have to be monitored. Monitoring is also required for certain safety-critical symbols defined in [ISO10, ISO 2575].

R7.6 – Software Fault Tolerance: [ISO08, ISO 15408-2] requires the detection of defined failures or service discontinuities and a recovery to return to a consistent and secure state (14.7.8.1) by using automated procedures (14.7.9.2). A list of potential failures and service discontinuities have to be supervised by a *watchdog* to detect entering of failure states. Furthermore, for a defined subset of functions that are required to complete successfully, failure scenarios shall be specified that ensure recovery (14.7.11.1).

R7.7 – System Integrity: In case of unrecoverable failures, the system shall be able to switch to *degraded operation mode* to preserve system integrity. A list of failure types shall be defined, for which no disturbance of the operation of the system can take place [ISO08, ISO 15408-2] (15.1.7.1). Moreover, the system shall ensure the operation of a set of capabilities for predefined failure types (15.1.6.1). This includes the handling of DoS attacks and detection of illegitimate policy changes. Some events have to be maintained in an internal representation to indicate if any violations take or took place. This includes the behavior of system activities for the identification of potential violations (7.3.10.2-3) like state changes (7.3.10.1).

2.2. Architecture

In this section, we briefly describe the architecture of a VAGS that addresses the identified requirements and is depicted in Fig. 2.1. While Certifiability (R6) applies to the complete development process, all other requirements can be fulfilled by the functionalities of the components of our architecture.

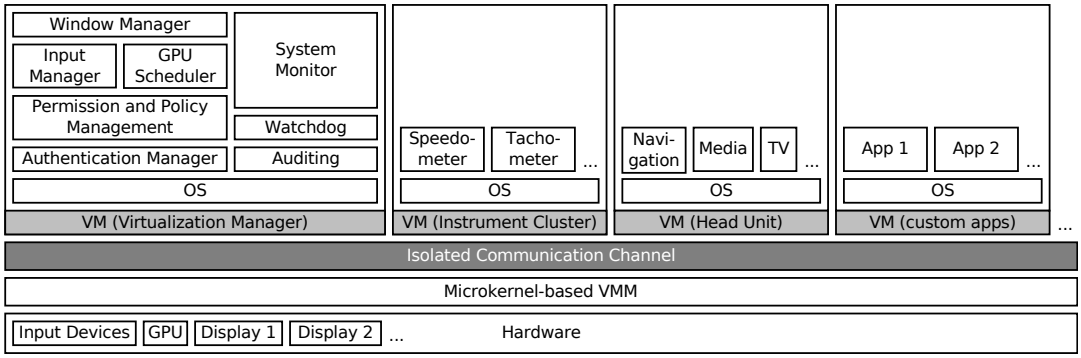


Figure 2.1.: Architecture of a virtualized vehicular graphics system

2.2.1. Virtualization

The consolidation of graphics hardware is of high relevance in modern cars. An increasing number of automotive functionalities and applications require highly sophisticated graphical representations in 2D or 3D based on hardware acceleration. For instance, the HU uses displays integrated into the backside of the front seats and center console to display multimedia content; and displays connected to the IC show car specific information like current vehicle speed or warnings. To this end, HU and IC both require a high amount of CPU and GPU resources, which makes them good candidates for hardware consolidation. Each virtualized ECU runs in a dedicated virtual machine (VM), and a virtual machine monitor (VMM) acts as middleware between VMs and hardware. Besides the already mentioned general benefits, the virtualization of IC and HU provides advantages such as the flexible placement of graphical output on previously separated displays, which is a matter of software implementation only. Moreover, virtualization enables OEMs to deploy custom applications inside a dedicated VM that is isolated from HU and IC.

With respect to certifiability, we follow the approach of a microkernel-based VMM where drivers run in user space rather than kernel space. Therefore, the kernel code size is very small and easier to certify [EJ09]. If driver code crashes,

this does not affect the VMM. The *Virtualization Manager* runs in a dedicated VM and exclusively manages shared resources. It contains relevant drivers, e.g., for GPU and input devices. This ensures that access to all shared resources is controlled by a single trustworthy VM. Indirect hardware access by VMs facilitates Virtualized Graphics Rendering (R4) and System Monitoring (R7). Additionally, the Virtualization Manager contains multiple software components ensuring that every hardware access by VMs is in compliance with our requirements. Note that our architecture only shows four exemplarily VMs. However, we do not restrict the number of VMs. Therefore, it is possible to deploy additional VMs if needed.

2.2.2. Inter-VM communication

In order to access hardware, the HU and IC VMs communicate with the Virtualization Manager VM. For this bidirectional communication, a Trusted Channel (R3) is required to support secure communication between the different virtual machines. A trusted channel is provided by the cooperation of the *Isolated Communication Channel* and the *Authentication Manager*. The Isolated Communication Channel provides integrity and isolation for communication (R3.1) between applications and the Virtualization Manager. To initiate a connection, applications first have to provide valid credentials to the Authentication Manager, to guarantee non-repudiation of origin and receipt (R3.2). In particular, this is required for the communication between the graphical applications located on HU or IC and the virtualization manager, which needs to be trustworthy to ensure that the active policy is never violated.

2.2.3. Integrity

In order to guarantee *Secure Boot* (R7.1), the integrity of code that is loaded must be verified, using, for instance, approaches described in [KXG12, GM08]. The *Auditing* component (R7.2) traces all relevant system activities and interactions. The gathered traces can be used by the Watchdog and System Monitor components to detect inconsistencies (for R7.3 to R7.7). The *Watchdog* supervises relevant system functionalities and emits signals in case of system malfunctioning as required for R7.3 to R7.6. The *System Monitor* receives signals of detected system malfunctions from the Watchdog. Rules are used to configure its reaction on these signals.

2.2.4. Application interfaces

Permission and Policy Management (R5) ensures that applications are getting their defined permissions to use functionalities or resources provided by the Input Manager, Window Manager, or GPU Scheduler. Permissions are represented by the active policy, which depends on the current state (R5.1), e.g., "vehicle is parking" or "vehicle is in motion". The policy management is configured by rules that define transitions between policies performed whenever state changes (R5.2) in defined time constraints (R5.3).

The *Input Manager* performs Input Event Handling (R1) and is responsible for dispatching user input events to the intended applications (R1.1). Since the processing of user input is subject to time restrictions, a minimal delivery time for input events to the applications must be ensured (R1.2).

The *Window Manager* provides the functionality for creating, positioning, and displaying windows of graphical applications. This represents a paradigm shift from fully user-defined window management to restricted window creation and positioning (R2). Applications with sufficient permissions interact with the Window Manager to create windows and to modify properties like size and position (R2.1). Moreover, the Window Manager is responsible for correct window stacking (R2.2) and meeting rendering time requirements (R2.3).

2.2.5. GPU Scheduler

The *GPU Scheduler* is responsible for Virtualized Graphics Rendering (R4) according to drawing requirements and permissions of graphical applications. To this end, applications are assigned priorities that define the amount of dedicated GPU resources (R4.1). Besides priorities, according to (R4.2), deadlines apply to the graphical rendering of certain applications like the tachometer. The GPU scheduler, therefore, has to sequence graphics commands, schedule application requests, and provide isolation between different contexts (R4.3).

2.3. Demonstrator

Within the ARAMiS project (cf., Sec. 1.3), multiple demonstrators were built for the domains avionic, railway, and automotive. For the automotive domain, two Virtual Car Telematics (VCT) demonstrators show concepts and possibilities of HU and IC consolidated on a single hardware platform. The VCT platform A (VCT-A) is based on a BMW vehicle and an Intel i7 platform, the VCT platform B (VCT-B) is based on a Daimler cockpit and a Freescale i.MX6 platform. Both VCT platforms focus on HMI on a virtualized platform. While the focus of VCT-A is security and using available hardware acceleration, the focus of VCT-B is safe 3D rendering on a shared GPU and safe, flexible display sharing. Thus, VCT-B demonstrates the feasibility to build a VAGS and relevant use cases. In this chapter, the VCT-B platform, related use cases, and evaluations are presented.

2.3.1. Hardware overview

In Fig. 2.2, the front of the VCT-B cockpit demonstrator is depicted. The

Figure 2.2.: Demonstrator front view with HMI devices

demonstrator has two automotive 12″ displays with a resolution of 1440 by 540 pixels. They represent the instrument cluster display and the head unit display known from today's high-end cars. User input is received from the steering wheel buttons and the push-and-rotary switch. The main implementation is running on a Freescale i.mx6 SABRE automotive infotainment platform, which features four ARM7 CPU cores running at 800 MHz, 2 GiB of RAM, a Vivante GC2000 3D GPU, and a Vivante GC320 2D GPU. The automotive displays are connected via LVDS to the Freescale platform. Since the LVDS connectors of

the platform are not compatible, adapters are used that connect one display to the HDMI port of the Freescale platform and the other to one of its LVDS ports. Additionally, the displays are connected to a CAN bus that is used to switch the displays on and off. An automotive rear-view camera with a resolution of 640 by 540 is connected via MIPI-CSI2. The steering wheel buttons and the push-and-rotary switch are connected via CAN to a Raspberry Pi that forwards the input events via Ethernet to the Freescale platform. For the sake of monitoring the demonstrator state and presenting the use cases, a HTML5-based GUI was developed, which runs on a separate Raspberry Pi and is accessed by a web browser using Wi-Fi. The GUI shows the *frame rate* of each application and whether it is active. Additionally, it allows to trigger the scenarios of our evaluation (cf., Sec 2.3.3). Fig. 2.3 shows a GUI screenshot.

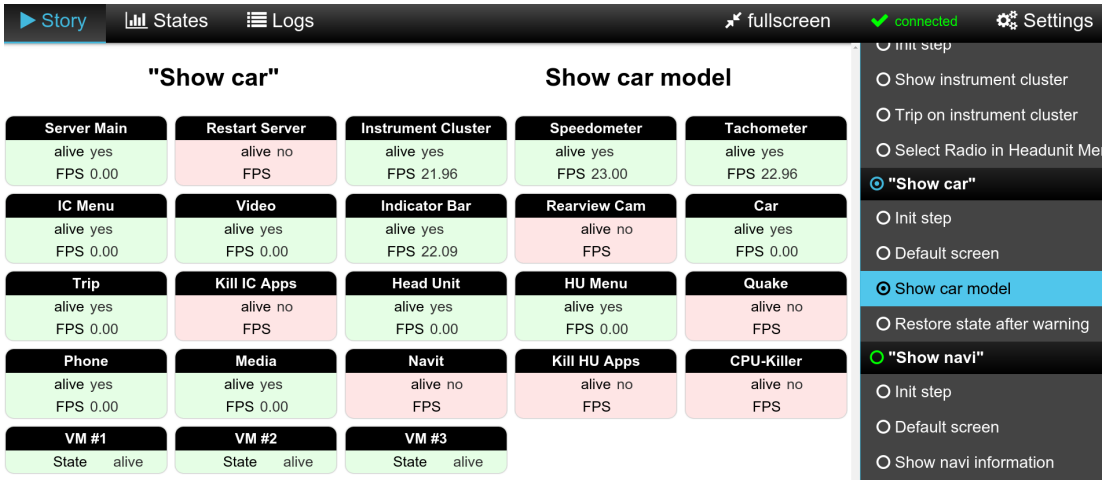


Figure 2.3.: HTML5-based demonstrator control GUI

2.3.2. Implementation

As software platform for the Freescale platform, the demonstrator uses a PikeOS 3.3 microkernel hypervisor [Pik16]. On top of the hypervisor, three paravirtualized Linux VMs are running, namely the Virtualization Manager VM, the Instrument Cluster VM, and the Head Unit VM (cf., Fig. 2.1). The Virtualization Manager VM has exclusive access to the GPUs and runs the GPU drivers. For inter-VM communication, we configured a couple of memory segments as shared between the VMs and use them for data transfer. We implemented stream-based bidirectional communication protocol using ring-buffers, which uses a small kernel module that allows a permitted process to map the shared memory into its memory address space. Inside the Head Unit

VM and the Instrument Cluster VM, representative 3D applications such as speedometer, tachometer, trip control, navigation, address book, and the game Quake 3, are executed. They connect to stub libraries for a window manager API, EGL, and OpenGL ES 2.0, to obtain a display access permission, create a window, and perform rendering, respectively. The stubs communicate to the Virtualization Manager VM, which performs display access control and safe sharing of the 3D GPU.

2.3.3. Evaluation

To demonstrate the use cases of a VAGS, multiple scenarios that contain relevant use cases were created. The scenarios were presented at the final ARAMiS event 2015 in Hamburg. In Fig. 2.4, a picture of the setup is depicted. The main demonstrator was the VCT-B cockpit demonstrator depicted on the right side. On the left side, we additionally presented the effectivity of 3D GPU scheduling using three displays connected to the same Freescale platform as used in the VCT-B. In this section, we briefly describe three representative scenarios that

Figure 2.4.: Setup of VCT-B (right) and GPU scheduling (left), at final ARAMiS event

show flexible HMI usage, isolation provided by the hypervisor, and the GPU scheduler effectivity, respectively.

Scenario 1

Initial situation: speedometer, tachometer, and IC menu displayed on IC display.

Action 1: the user selects “Contacts” in the IC menu.

Situation: contacts are displayed on the IC display.

Action 2: the user uses IC menu to use also the HU display.

Situation: a contact is additionally displayed on the IC display.

Scenario 1 demonstrates that the two connected displays can be flexibly used by all applications that were granted access. The limitations of the existing approach, which uses physically separated hardware platforms for IC and HU, no longer apply.

Scenario 2

Initial situation: speedometer and tachometer on IC display, radio on HU display.

Action 1: the user starts game Quake 3 using the HU menu.

Situation: the game Quake 3 is started and displayed on the HU display.

Action 2: the Head Unit VM (running Quake 3) crashes.

Situation: the Head Unit VM and Quake 3 no longer work, the IC is unaffected.

Scenario 2 demonstrates that even a crash of one of the VMs and the applications running inside does not affect the operation of Virtualization Manager VM and the Instrument Cluster VM. The hypervisor effectively isolates the VMs.

Scenario 3

Initial situation: no application is running, displays are blank.

Action 1: the speedometer is started with highest priority on display 1.

Situation: the speedometer runs at 60 FPS on display 1.

Action 2: two instances of Glmark2 [glm] with lower priority are started.

Situation: the rendering of the speedometer is not affected. On the displays 2 and 3, the Glmark2 instances are running. Since they have different priorities, only the lowest priority instance is affected from high stuttering.

Scenario 3 demonstrates that our 3D GPU scheduler is able to guarantee the *frame rate* for the speedometer application, which has the highest priority. Applications with lower priority only get the available remainder of GPU resources that can be used without affecting the rendering of higher priority applications.

2.4. Related Work

The concept of microkernel-based VMMs in virtualization is well known for many years. The focus on safety increased during the last few years, e.g., the NOVA microkernel [SK10]. Moreover, certifiability became more important, at least in case of the VMM [KAE+10].

A large number of work related to virtualization and graphics applications has been described in the literature. In this section, we present related work on windowing systems and graphics forwarding, while related work for execution time prediction and GPU scheduling is presented in Sec. 3.9 and Sec. 4.7. According to [JE91], the X11 Windowing System does not provide security. Trusted X [EMP+91] has been proposed to provide security for the X Windowing System targeting the requirements in TCSEC B3 (superseded by [ISO08, ISO 15408-2]) but has not been certified. To provide isolation, an untrusted X server and a window manager is deployed for each security level, which impacts scalability. Therefore, mutual isolation of applications is practically impossible due to scalability issues. Nitpicker [FH05] is a GUI server with security mechanisms and protocols to provide secure and isolated user interaction using different operating systems. To achieve isolation between these OSes, Nitpicker uses the VMM L4/Fiasco [Hoh02]. The EROS Window System (EWS) [SVNC04] targets the protection of sensitive information and the enforcement of security policies by providing access control mechanisms and enforcing the user volition. A common denominator of Trusted X, Nitpicker, and EWS is that they only focus on security and thus do not comply with Input Event Handling (R1), Restricted Window Creation and Positioning (R2), and System Monitoring (R5). DOpE [FH03] is a window server that assures redrawing rates windows of real-time applications and provides a best-effort service for non-real-time applications. DOpE is based on L4/Fiasco [Hoh02] for isolation and IPC. However, policies are not enforced. Common to all these windowing systems is the fact that they do not support graphics hardware acceleration and do not provide any timing guarantees for rendering and displaying.

VMGL [LCTSdL07] is an approach to transfer OpenGL commands from an OpenGL client to an OpenGL server using a TCP/IP connection. However, using TCP/IP causes significant latency and overhead. Xen3D [Smo09] uses the MESA open source graphics framework and executes only part of the graphics stack on the Virtualization Manager (called “Dom0” in XEN). Shared memory

communication is used to transfer MESA-internal data between the VMs. Unfortunately, a concept like Xen3D only works if the user space driver is open source. The VMware hosted architecture [DS09] uses an emulated GPU architecture, which uses the physical GPU connected to the Virtualization Manager to achieve fast rendering. This approach has the advantage that also proprietary GPU drivers can be used but needs additional overhead for translating between the emulated and the physical GPU architecture. GViM [GGS+09] uses a similar approach for GPGPU based on CUDA.

Approaches with mediated pass-through [TDC14, Int16] use the interface between user space driver and the operating system kernel. The Virtualization Manager therefore receives the batches of GPU opcode and is not fully aware about what code is going to be executed on the GPU. Mediated pass-through is popular since its overhead is quite low and a good performance can be achieved. While such a concept provides good isolation if the user space driver crashes, it does not prevent the creation of malicious GPU opcode, which could crash the GPU or break isolation between VMs.

Blink [Han07] is a display system that focuses on the safe multiplexing of OpenGL programs in different VMs. Blink uses an OpenGL Client/Server to transmit the OpenGL commands and data via shared memory to a "Driver VM". The "Driver VM" is responsible for the execution of the OpenGL commands on the GPU. Blink proposes "BlinkGL", which increases performance, but requires applications to be modified. In our automotive cockpit demonstrator VCT-B described in Sec. 2.3, we used the concepts of [Eis14], which use isolated shared memory communication to efficiently transfer OpenGL ES 2.0 commands between VMs.

2.5. Summary and Appraisal

In this section we presented the requirements for automotive HMI systems. From ISO standards, automotive design guidelines, and OEM-specific demands, we derived seven technical requirements R1 to R7. Our system model is based on a virtualized system to consolidate HU and IC on a single hardware platform. We proposed a VAGS, which contains the components needed to fulfill R1 to R7.

To be compliant with legal, safety, and OEM requirements, a VAGS must fulfill many requirements. Many parts of a VAGS can already be developed using state-of-the-art concepts. For instance, microkernel-based hypervisors like [Pik16, Int15] target automotive scenarios and thus can be used for a VAGS. The Authentication Manager, the Watchdog, and the System Monitor can be implemented by automotive software developers according to ISO 26262 [ISO11]. Similarly, for the Input Manager existing concepts can be extended. The major conceptual challenges of a VAGS are the Window Manager and the GPU Scheduler.

Although window managers are very common in graphical user interfaces, the existing concepts assume that the user shall be in control of the system and that the task of a window manager is primarily to flexibly adapt to the user's desires. However, in the automotive domain, the visibility of many functions must be guaranteed and even be consistent. To this end, the compliance of a window manager with the requirements must be inherently provided by the window manager itself. This necessitates context-based access control mechanisms for safe display sharing. Safe display sharing—and thus the Window Manager—are not the focus of this work but are presented in [GSGH+14, GSGH+15, Gan17].

Our automotive cockpit demonstrator VCT-B shows the feasibility of a VAGS—the motivated goal to reduce cost, installation space, and energy consumption is achievable. Additionally, it demonstrates that the newly-gained flexibility in display usage allows for next-generation HMI systems.

GPU scheduling is challenging in both, conceptual and technical terms. A rendering scene is composed of a sequence of frames. In order to guarantee the rendering of important 3D applications and ensure a smooth animation without stuttering, a real-time 3D GPU scheduler with frame-individual deadlines is needed. Additionally, 3D GPUs do not support preemption with a guaranteed maximum latency. The GPU scheduler is the main focus of this work and consists of an execution time prediction and a GPU scheduling framework, presented in Chapter 3 and Chapter 4, respectively.

3. Execution Time Prediction

Applications that use the GPU for 3D rendering submit GPU command batches. The GPU executes these batches one at a time. In Sec. 1.2.3, we presented the goals and the problem statement for the execution time prediction of 3D GPU command batches. In this chapter, we present our corresponding concepts for the prediction of the execution time of 3D GPU command batches. We propose a framework that measures and predicts the execution time on the GPU for 3D applications that do not have to be modified. To this end, we perform prediction in the user space, using only the OpenGL API as interface. We present multiple heuristics and compare our results with the measured GPU execution times. According to our evaluation results our concepts provide a good estimate for the real GPU execution time.

This chapter is structured as follows. In Sec. 3.1, we present the background: the used graphics APIs, machine learning, and model analysis. The system model is described in Sec. 3.2, followed by the architecture with the components of our prediction concepts in Sec. 3.3. The prediction models for the GPU commands Flush, Clear, and SwapBuffers are presented in Sec. 3.4. The models for the challenging GPU command Draw command are presented in Sec. 3.5, which uses multiple submodels. The submodels to estimate the number of fragments are described in Sec. 3.5.1. To estimate the execution time of shader programs, a profiling-based approach is presented in Sec. 3.5.2 and an approach based on machine learning is described in Sec. 3.5.3. Additionally, we propose an optional concept for online adaption that is explained in Sec. 3.6. In Sec. 3.7, our implementation is described. In Sec. 3.8, we evaluate the prediction accuracy and conclude with related work in Sec. 3.9 and a summary in Sec. 3.10.

3.1. Background

This section contains background information on 3D rendering and machine learning. When an application uses the GPU for 3D rendering, it uses standardized graphics APIs. As a vendor-independent consortium, the Khronos Group [Khrb] publishes open standards for GPUs, including—but not limited to—use cases for embedded systems. We provide an overview about EGL, which connects to the native GPU platform, and OpenGL ES 2.0, which contains the actual rendering commands.

3.1.1. EGL

The EGL API [Lee14] published by Khronos [Khrb] is commonly used to connect rendering with the underlying native windowing system. To allow for rendering, the following steps are performed using EGL. First, references to the native display and native window are obtained, using, e.g., "XOpenDisplay" and "XCreateWindow" on a X11-based platform, or "fbGetDisplayByIndex" and "fbCreateWindow" on a framebuffer-based Freescale platform. Second, an EGL context is initialized and then activated with "eglMakeCurrent", using an EGLDisplay (obtained using a native display) and an EGLSurface (obtained using a native window). In this step, multiple options can be used, for instance to define the color and depth buffer formats. As soon as a context is active, the OpenGL ES 2.0 can be used for rendering. The selected EGL context can be changed using "eglMakeCurrent", which determines the render buffer and acts as a scope for OpenGL. This means that all OpenGL calls affect the current EGL context, only[1]. Eventually, if a frame has been rendered completely, the content must be made visible at the selected window using "eglSwapBuffers". To this end, the call "eglSwapBuffers" manifests the EGL surface buffer to the associated native window. If applicable, the window manager's compositor is notified and typically copies the content of the native window to the respective position on a connected display.

3.1.2. OpenGL ES 2.0

The OpenGL ES 2.0 API [ML10] is widely used for 3D rendering on embedded devices such as smartphones, automotive head units, and instrument clusters. The latest version of OpenGL ES is 3.1 and is backward compatible down to

[1]Optionally, data sharing can be enabled, though.

version 2.0. Many applications, especially those with publicly available source code, do not use features beyond OpenGL ES 2.0.

An application that wants to perform 3D rendering, first initializes and activates an EGLContext (see previous section). Second, it initializes 3D rendering, for instance by activating desired features, loading 3D models into memory, and creating an OpenGL program. Third, it starts rendering by submitting one or more DRAW commands, typically in a so-called rendering loop where each iteration generates one frame. Inside the loop, typically "glClear" is used to reset the render target to the background color. Subsequently, DRAW commands are issued and eventually, "eglSwapBuffers" makes the rendered frame visible, typically using a Window Manager. All OpenGL ES 2.0 API calls refer to the scope of the current OpenGL ES 2.0 Context, which is created together with each EGL context.

The actual rendering tasks are submitted via DRAW calls, which follow multiple conceptual[2] in Fig. 3.1. The first step is the vertex shader, which is

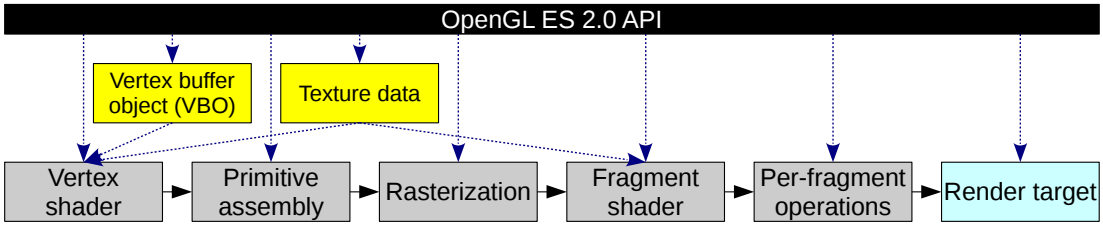


Figure 3.1.: OpenGL ES 2.0 rendering pipeline

code written in GLSL [Sim09] and compiled by the user space driver for the used GPU architecture. The input data of a vertex shader consists of constant (as per DRAW call) parameters called uniform variables and per-vertex input data called vertex attributes. The data for the vertex attributes can either be provided as a simple pointer to the vertex array data, or a vertex buffer object (VBO) can be created. Using a VBO can reduce the number of OpenGL ES 2.0 calls and the overhead of copying the vertex attributes to GPU memory (if present). The output data of a vertex shader is the vertex position and (optionally) so-called varying variables, i.e., user-defined input parameters for the fragment shader. The vertex position is assigned to the special variable "gl_Position". Next, we briefly describe the typical approach how this calculation is performed [GPSM14, Ope11]. The input vertex coordinates are typically in object (or model) coordinates, i.e., all coordinates are given relative

[2]However, driver or GPU are allowed to switch order if the result is not affected.

to the 3D object's center. Typically, such objects are rotated, translated (i.e., relocated), and scaled. These three operations can be combined into a model matrix such that multiplying the input vertex coordinates with the model matrix performs the three operations and produces the world coordinates. The world coordinates are multiplied with the view matrix to produce eye coordinates. Figuratively, the view matrix moves the camera to the desired position and direction. Since the model matrix and the view matrix are mathematically based on the same operations, they are often combined to a model view matrix. The last step is the transformation to clip coordinates using the projection matrix. This represents the perspective, i.e., what and how the camera looks on the scene, such as the viewing angle and the aspect ratio. Again, the projection matrix can be combined with the model view matrix to a model view projection matrix (MVPM). In order to achieve fast vertex shader execution, it is common to use a MVPM calculated by the CPU and passed to the GPU as a uniform variable. The primitive assembly step creates base primitives, for instance a triangle strip is split up into individual triangles. Additionally, if "culling" is used, back-facing triangles are omitted. Rasterization converts the base primitives to a two-dimensional set of fragments (possible pixels) depending on resolution and size of the render buffer and the viewport. The fragment shader is executed for each fragment and calculates its color. For instance, texture data can be aligned to the vertex grid in order to create realistic-looking scenes. Textures are special kinds of uniform variables holding image data, which is separately uploaded using API calls such as "glTexImage2D". According to the OpenGL ES 2.0 API, an OpenGL program represents the compiled and linked combination of both, vertex shader code and fragment shader code. To perform a Draw call, first the OpenGL program must be selected by "glUseProgram". To use multiple OpenGL programs, an application can therefore switch them between the Draw calls. The OpenGL programs are typically not created in the rendering loop but before while a scene is loaded. The per-fragment operations contain further steps. For instance, the scissor test allows to define a rectangular area outside of which no fragments are drawn. If enabled, the depth test compares the current depth value of a pixel with the depth value of the fragment to determine if the fragment should be skipped. Since OpenGL ES 2.0 does not allow the fragment shader to change the depth of a fragment, the GPU can increase the execution speed by performing the depth test (a per-fragment operation) before the fragment shader—an optimization called early depth test). Blending allows to

combine the color of a pixel with the color value of the fragment, e.g., to make objects look transparently. If a fragment has passed the per-fragment operations, the color value and (optionally) the depth value of the respective pixel on the render target is updated. The render target is often the EGL surface buffer, which causes the GPU to write to the associated native window. However, it is also possible to use textures as render targets, which allows for sophisticated 3D effects.

3.1.3. Machine Learning

Concepts for machine learning automatically learn programs from data [Dom12]. We distinguish between supervised and unsupervised learning. For supervised learning, the machine learning knows which values are input and which values are output, for unsupervised it does not [HTF09]. In this work, we only focus on supervised learning. The elements of the input data used for machine learning are called *features*. A feature can be a measurement or preset value or can be calculated from one or more of the measured values. Machine learning uses the input data—denoted as X—to predict the output data—denoted as Y. In general, X is a column vector that contains p input values.

$$X = \begin{bmatrix} 1 \\ X_1 \\ \vdots \\ X_p \end{bmatrix}$$

The term 1 is included in X to support an intercept (also called bias). While Y in general also is a column vector, we focus on Y as a single value. Next, we briefly introduce linear regression, MARS, and artificial neural networks, based on [HTF09].

3.1.3.1. Linear Regression

A linear model assumes that Y linearly depends on X. For an input vector $X^T = (1, X_1, ..., X_p)$ the output Y is predicted by $\hat{Y} = X^T\beta$. This means that each element of the input vector X is multiplied by the corresponding weight in β. To fit a linear model to a set of training data, typically *least squares* are used. Using this method, β is selected such that the residual sum of squares (RSS) is minimal, with $RSS(\beta) = (\widetilde{Y} - \widetilde{X}\beta)^T(\widetilde{Y} - \widetilde{X}\beta)$. In this formula, $\widetilde{X}$ and $\widetilde{Y}$ represent the training data. The $(N \times (p+1))$ matrix $\widetilde{X}$ contains N samples of input data and the $N \times 1$ matrix $\widetilde{Y}$ contains the corresponding measured output.

Linear regression also supports non-linearity, if non-linear features, e.g., polynomial features, are used. Another option to deal with non-linearity are continuous piecewise features. With continuous piecewise features, the range of $\widetilde{X}$ is divided into continuous intervals. Each interval has one or more associated features used to create a model where the transition between adjoint intervals is continuous. In Fig. 3.2, an example is provided where a continuous piecewise

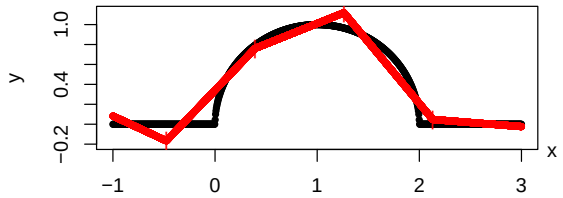


Figure 3.2.: Example: Continuous piecewise linear regression model fitting half circle

linear model with a constant interval size of 1 was fitted to training data of a half sphere. Although the training data is obviously non-linear, the linear models of the individual intervals provide a reasonable approximation.

3.1.3.2. MARS (Multivariate Adaptive Regression Splines)

MARS [Fri91] is a regression-based concept that creates the needed functional relationship between X and Y automatically. To this end, MARS generalizes continuous piecewise linear regression by determining good intervals and the relevant features and feature combinations for each interval. In more detail, MARS uses the training data $\widetilde{X}$ to create a model $M = \beta_0 + \sum_{i=1}^{k} (c_i \times F_i)$, with $F_i = \beta_i \times \prod_{y \in Y \subseteq \mathcal{P}} MAX(0, e_{y,i})$ with $|Y|$ being called degree of the term F_i. The expression $e_{y,i}$ is either of the form $(X_y - c_{y,i})$ or $(c_{y,i} - X_y)$. Each MARS model contains exactly one term of degree 0, i.e., β_0. When choosing the coefficients β_i, MARS uses linear regression by minimizing the RSS. Implementations of MARS typically allows customization through parameters. Important parameters (for the "earth" implementation in R [Mil11]) are:

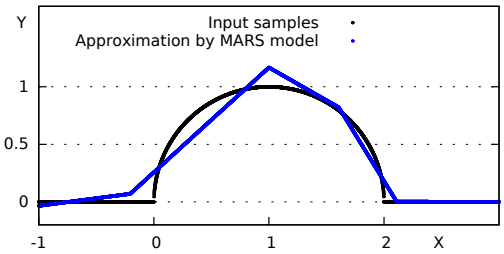
degree: The maximum degree allowed for the terms. While a higher degree could improve accuracy, if chosen too high, the model could be overfitting and become very huge.

thresh: A model with more terms is preferred, if it improves the estimation of the training data by at least the "thresh" value.

nk: Maximum number of terms allowed. This stops searching for a better model, if *nk* terms were found.

fast.k: Boolean value that determines whether for selecting terms a heuristic shall be used. Using the heuristic can speed up the runtime for the MARS algorithm, but the determined model can be worse than without using it.

Fig. 3.3 shows an example where input data following a half circle was created and fitted by a MARS model using the implementation [Mil11]. The model



(a) Example: Input values and fitted MARS model for half circle

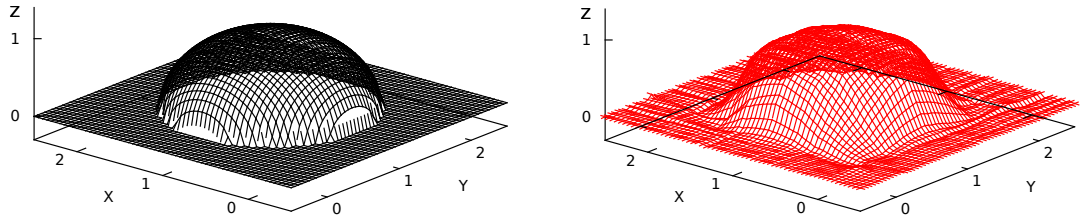
	coefficients
(Intercept)	0.2200212
h(x- -0.2097)	0.7834277
h(0.9999-x)	-0.1267038
h(x-0.9999)	-1.3526708
h(x-1.6019)	-1.0412111
h(x-2.1143)	1.6108366

(b) Example: MARS model fitted for half circle

Figure 3.3.: Error of model for auxiliary fragment shader execution time

has some deviation but needs only five terms plus intercept (β_0). If desired, the optimization threshold can be specified, e.g., a lower threshold might result in more terms but lower deviation. The function MAX(0, x) is called h(x), here. Since this example contains only one feature, MARS behaves similarly to continuous piecewise linear regression with automatically calculated intervals. It tries to use as few intervals as possible to achieve the desired accuracy (specified by the threshold).

An example with two-dimensional dependent input data is provided in Fig. 3.4, where a half sphere is used as input data. MARS was used with a low threshold, thus more accurately matching the input data. As a result, the



(a) Example: Input values for MARS (half sphere)

(b) Example: Values fitted by MARS model

Figure 3.4.: Example: Input values and MARS model for half sphere

MARS model is very accurate and shows that MARS can adapt to multi-dimensional non-linear data. The MARS models contain many terms of degree two that were automatically calculated based on the training data. If a similar expressiveness would have to be achieved with continuous piecewise linear regression with constant intervals, a two-dimensional grid of the feature space $x \times y$ has to be created with sufficiently small grid segments. Clearly, continuous piecewise linear regression does not scale well with the number of non-independent features. Therefore, if a model for data is needed that contains multiple non-independent features, MARS is a much better choice. In this case, a MARS model can be orders of magnitudes smaller than a continuous piecewise linear regression with constant intervals that has similar accuracy.

3.1.3.3. ANNs (Artificial Neural Networks)

The term Artificial Neural Network (ANN) is used for many different model concepts. Often, ANNs are used for classification, but also regression is common. In this section, we focus on the widely-used feed-forward regression networks with a single output unit. Note, that only a very brief introduction is presented that is sufficient to understand the reasoning in this work. An ANN can be represented by a network graph, such as the example in Fig. 3.5. This example contains three

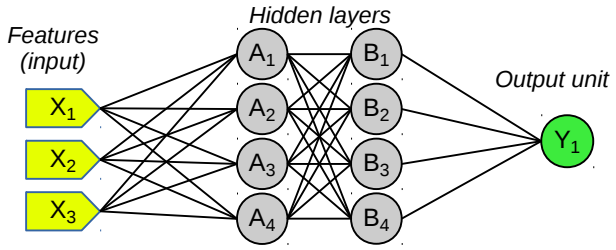


Figure 3.5.: Example of a feed-forward artificial neural network graph

input features, two hidden layers of size 4, and a single output unit. The nodes in the graph are called neurons and use internal weights to compute their output value from their respective input data. A simple approach uses linear regression, as described in Sec. 3.1.3.1. However, an ANN is more expressive since multiple nodes (typically very many) are used in each layer and multiple layers can be concatenated in such a way that the output of one layer serves as input to the next layer. Additionally, ANNs use an activation function that is applied to the weighted input. Common activation functions include the sigmoid $\sigma(v) = \frac{1}{1+e^{-v}}$, $tanh$, and the Rectified Linear Unit $ReLU(x) = max(0, x)$. Activations functions typically introduce non-linearity. Although complex linear models might be able

to fit non-linear data to some extent, introducing non-linearity through activation functions typically provides better accuracy at a higher efficiency (since a smaller network graph can be used). Generally speaking, ANNs can provide a very high expressiveness and accuracy but have the disadvantage that a high number of weights is used, which requires a high amount of training data and significant computation time for prediction. Moreover, the network graph and the used types of neurons heavily affect the suitability of an ANN for a certain task and therefore must be well-chosen.

3.1.4. Model analysis

Models such as linear regression, MARS, and ANNs, are based on training data. It is therefore important to measure the accuracy of the mapping. In this section, two common plot types are introduced, which can be used to determine the accuracy of a model. The goal of a model is an accurate estimation y for given input data. For the typical case, where the data contains jitter due to limited measurement accuracy or side-effects, perfect accuracy is impossible. Accuracy is therefore analyzed using statistical methods comparing a created model with a sufficiently large set of data samples. For each data sample, the y value predicted by the model is called *fitted* value. The measured y value is called *residual*. Next, we briefly explain two common statistical plots, which provide a visual overview of prediction accuracy.

Cumulative distribution plot. For each data sample, the absolute deviation between the residual and fitted values are calculated. In a cumulative distribution function, the deviation is laid on the x-axis. For each of x value, the proportion (percentage) of the data set with a deviation not exceeding the x value is depicted on the y-axis. For instance, a cumulative distribution plot shows that 95 % of the fitted values deviated not more than some value from the residuals.

Residuals vs. Fitted plot. For each sample, the deviation between the residual and fitted values is calculated. The fitted values are laid on the x-axis. For each fitted value, the deviation to the residual is drawn as a point at the respective height. A sample with a y-value of 0 means that the model perfectly fitted the residual. A y-value higher (lower) than 0 means that the residual was higher (lower) than the fitted, i.e., the model underestimated (overestimated) the residual.

3.2. System model

The components and interfaces of our system are depicted in Fig. 3.6. Basically, the system consists of four layers, namely, the application layer, the user space driver layer, the kernel space driver layer, and the hardware layer. For the sake of a concrete description and since for the evaluations in this work a variety of 3D applications is required, we consider the most common 3D rendering API for embedded systems, OpenGL ES 2.0 [Khra]. Our system model is state-of-the-art at recent embedded GPUs, for instance from Vivante, ARM, or Nvidia (cf., [SGDR14] (Figure 1) for the Nouveau driver [Nou]). On the

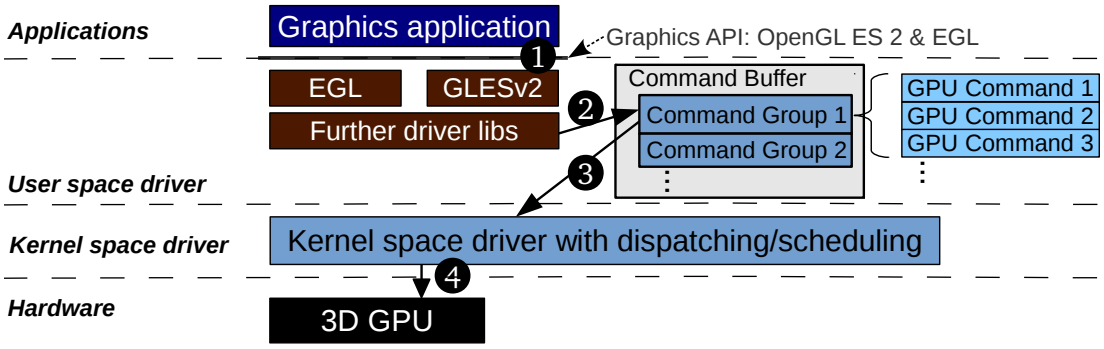


Figure 3.6.: Hardware and software components for 3D rendering with OpenGL ES 2.0

application layer, *graphics applications* use OpenGL ES 2.0 and EGL as interfaces (❶ in Fig. 3.6). Thus, graphics applications based on these common APIs do not have to be changed. From the OpenGL ES 2.0 commands, the *user space driver* creates GPU commands. Consecutive batches of GPU commands are called command groups (CGs) and are enqueued in the *Command Buffer*. The Command Buffer resides in shared memory accessible from user space and kernel space (❷ and ❸). Many drivers perform a system call to notify the *kernel space driver* of the added CG (❸). The system call can be intercepted by instrumenting the user space driver (if its source code is available) or by intercepting the "ioctl" function. The kernel space driver is responsible for dispatching the CGs to the GPU and thus determines the order of execution (❹). Once the GPU starts execution, the execution of CGs cannot be preempted.

3.3. Prediction Architecture

Fig. 3.7 shows an overview of our prediction framework architecture with its three basic components, namely, OpenGL ES Context Monitor (A), Predictor (B), and Execution Time Monitor (C), and their embedding into the overall system. The main component of our framework is the Predictor (B), which predicts the

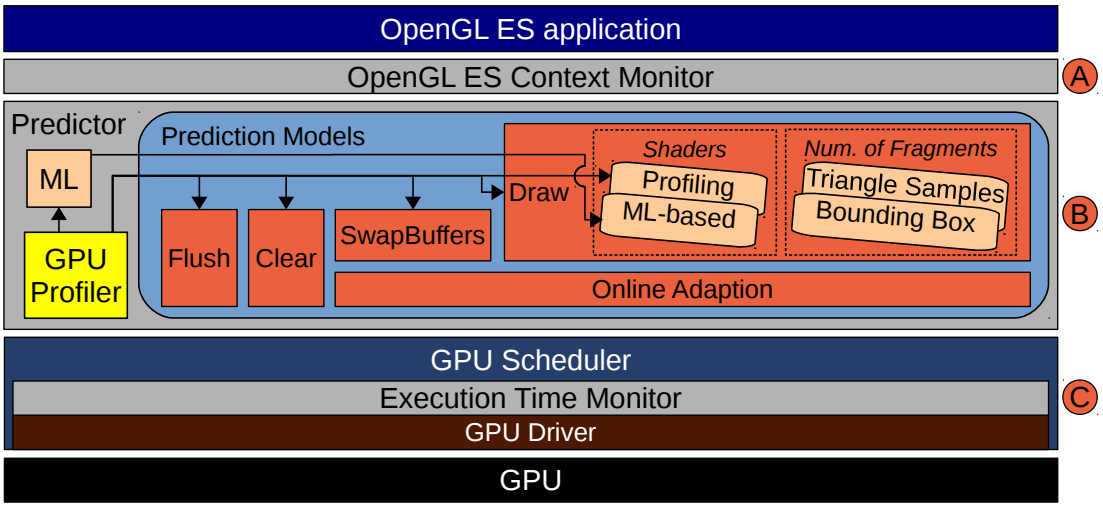


Figure 3.7.: Execution Time Prediction Components and Models

execution time of CGs depending on the OpenGL ES 2.0 Context. The prediction models are based on the measurements of the real execution time of CGs, which are provided by the Execution Time Monitor (C). Next, the three components of execution time prediction are explained in more detail.

3.3.1. OpenGL ES Context Monitor

The execution time of OpenGL ES 2.0 commands depends on their OpenGL ES 2.0 Context. For instance, the execution time of the DRAW command is context-dependent. As described in Sec. 3.1.2, a DRAW command uses the OpenGL rendering pipeline whose execution time depends on many parameters previously defined by OpenGL commands. The OpenGL ES Context Monitor of our framework intercepts all OpenGL ES 2.0 and EGL API calls to create and maintain a local copy of the relevant parameters of the OpenGL ES 2.0 Context. To this end, 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 ES 2.0 calls by the application. For instance, if the

application calls glBindBuffer, the OpenGL ES 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 ES 2.0 commands.

3.3.2. Predictor

The Predictor is the main component of our execution time prediction and uses several models to predict different types of GPU commands. It determines the execution time of the GPU commands based on the current OpenGL ES 2.0 Context and the prediction models. It is notified when calls to the OpenGL ES 2.0 functions of the GPU driver actually emit a CG. The predicted execution time of a CG is then calculated by summing up the execution times of the included GPU commands. It is attached to the CG and serves as input to a GPU scheduler.

3.3.2.1. GPU Profiler

The GPU Profiler determines the performance of the 3D GPU through profiling. To this end, we create suitable sequences of OpenGL commands, which are passed to the native driver and finally glFlush is called to submit all commands to the GPU. After that, we wait until the Execution Time Monitor detects that the GPU has finished execution. Subsequently, the measured execution times are read from the Execution Time Monitor. The GPU performance is then calculated based on multiple measurements using either the respective average or linear regression. The performance parameters can by classified into system-specific performance parameters and OpenGL program-dependent performance parameters. System-specific performance parameters depend on the specific hardware platform, such as GPU type, GPU speed, CPU speed, and memory speed. However, they do not depend on the OpenGL program used by an application. Therefore, it is sufficient to run the profiling benchmarks just once. The application-dependent performance parameters in our concept are used for the profiling-based shader model (cf., Sec. 3.5.2), where the GPU Profiler is profiling each OpenGL program. All other performance parameters are system-specific. The GPU Profiler persistently stores the determined performance parameters. This avoids re-running the GPU Profiler, if the knowledge is already available from an earlier run.

For the shader model that is based on machine learning, tailored 3D profiling applications are used, which execute a huge variety of shaders. The shader

parameters and the measured execution times are recorded by the Execution Time Monitor and used as training data to create machine learning models (cf., Sec. 3.5.3).

3.3.2.2. Prediction Models

A GPU has a hardware-specific instruction set. Many instructions change internal GPU state or set register variables and are so fast that their execution time cannot be measured and therefore are not predicted. For the remaining GPU commands—namely, FLUSH, CLEAR, SWAPBUFFERS, and DRAW—the Predictor uses the models shown in Fig. 3.7. The prediction models use the parameters provided by the GPU Profiler depicted in Table 3.1. The ML-based

Table 3.1.: Performance parameters provided by the GPU Profiler to the prediction models

Parameter	depends on	Used by
pc_{flush}	system	FLUSH model
$pc_{\text{clear[btypes]}}$	system	CLEAR model
$pc_{\text{swapbuffers}}$	system	SWAPBUFFERS model
$pc_{\text{drawconst}}$	system	Profiling-based shader model for DRAW
pc_{depth}	system	Profiling-based shader model for DRAW
pc_{blending}	system	Profiling-based shader model for DRAW
$pv_{\text{vertex\_shader}}$	OpenGL program	Profiling-based vertex shader model for DRAW
$pv_{\text{fragment\_shader}}$	OpenGL program	Profiling-based fragment shader model for DRAW

shader model is calculated using machine learning (ML), which provides and uses the models depicted in Table 3.2. The rather simple models for FLUSH

Table 3.2.: Machine learning models provided for shader prediction

ML model	depends on	Used by
m_{cmdsVS}	system	ML-based vertex shader model for DRAW
m_{auxVS}	system	ML-based vertex shader model for DRAW
m_{cmdsFS}	system	ML-based fragment shader model for DRAW
m_{auxFS}	system	ML-based fragment shader model for DRAW
m_{texld}	system	ML-based shader model for DRAW

(model m_{flush}), CLEAR (model m_{clear}), and SWAPBUFFERS (model $m_{\text{swapbuffers}}$) are described in Section 3.4. The challenging DRAW command (model m_{draw}) and its sub models to estimate fragments and shaders are presented in Section 3.5. The optional *Online Adaption* allows to correct predictions leaning to either overestimation or underestimation and is presented in Section 3.6.

Listing 3.1: **Execution time prediction for CGs**

```
1 on receive of a Flush:
2    et_flush += m_flush()
3 on receive of a Clear call:
4    et_clear += m_clear(btypes, s_rb)
5 on receive of a  call:
6    if coming from eglSwapBuffers:
7       et_sb += m_swapbuffers(s_rb)
8    else:
9       et_draw += m_swapbuffers(s_rb)
10 on receive of a Draw(vertexList) call:
11   vertices += vertexList
12 on change or end of a rendering scene:
13   et_draw += m_draw(n_Calls, vertices, ctx)
14   vertices = ∅
15 on submission of nextCG:
16   nextCG.pred_ET = online_adaption()
```

The Predictor uses these models to predict the execution time of CGs as described in Listing 3.1. The execution time for FLUSH commands—estimated by the model m_{flush}—is aggregated in the variable "et_flush" (Lines 1–2). Likewise, the variable "et_clear" aggregates the execution time for CLEAR commands (Lines 3–4), which is modeled by m_{clear}. The model $m_{\text{swapbuffers}}(s_{\text{rb}})$ is used for so-called buffer resolves (cf., Sec. 3.4.3). The typical case is a SwapBuffers command, but buffer resolves can also occur in the context of DRAW calls. The estimated execution time of the model $m_{\text{swapbuffers}}(s_{\text{rb}})$ is therefore aggregated either in the variable "et_draw", or in the variable "et_sb" (Lines 5–9). For the DRAW command, we perform the prediction that uses the model $m_{\text{draw}}(n_{Calls}, vertices, ctx)$ on batches of OpenGL DRAW calls. To this end, DRAW calls are not directly predicted, but the respective vertices are accumulated in the set "vertices" (Lines 10–11). For applications that use many OpenGL DRAW calls, this batch prediction reduces the CPU overhead introduced by execution time prediction significantly. DRAW call batches have to render the same rendering scene, e.g., they must share the same model view projection matrix, viewport, depth range, primitive mode, and GL capabilities. If the rendering scene changes or is ended by the imminent submission of a CG, the current batch is complete and gets predicted (Lines 12–14). When the CG is eventually about to be submitted to the kernel space, the function "online_adaption()" returns its predicted execution time (Lines 15–16). If online adaption—which is optional—is disabled, the function "online_adaption()" returns the sum of the four kinds of predicted execution times. If online adaption is enabled, history-based smoothing is applied (cf., Sec. 3.6).

3.3.3. Execution Time Monitor

To create and verify our prediction models, we use execution time measurements of CGs. To this end, we need to know when the GPU execution of a CG starts and when it has finished. Doing these measurements in user space would be inaccurate if the CPU is loaded, since any delay in context switches would affect measurement accuracy. Therefore, the Execution Time Monitor takes the timestamps as close to the GPU as possible, i.e., in the kernel space driver.

In order to detect in kernel space when a CG has finished, we use a similar technique as described in [KLRI11]: we let the GPU acknowledge the execution of each CG. To this end, we append a GPU instruction to each CG that makes the GPU create an interrupt each time a CG has finished execution. Additionally, we take the timestamp when dispatching of a CG is finished, i.e., directly before the GPU can start executing it. If the GPU is idle, the execution time is the interval between this timestamp and the time of the respective interrupt request. If the GPU is not idle, the time interval between two consecutive interrupt requests is used as execution time.

3.4. Prediction models for Flush, Clear, and Swapbuffers

3.4.1. Prediction Model for Flush

A Flush is a special case, since it represents the constant time needed for the execution of a CG on the GPU. The emission of a CG can be caused by multiple API calls, e.g., glFlush, glFinish, or eglMakeCurrent. Since Flush 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 for the Flush command can be estimated by the following simple model $m_{\text{flush}}()$, where pc_{flush} is a system-specific constant provided by the GPU Profiler.

$$m_{\text{flush}}() = pc_{\text{flush}} \tag{3.1}$$

3.4.2. Prediction Model for Clear

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, and stencil buffer, should be cleared. As shown in [GSC+15], the execution time to clear a buffer linearly depends on the render buffer size[3] 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. Thus, the predicted time to clear a certain set of buffers can be modeled by

$$m_{\text{clear}}(btypes, s_{\text{rb}}) = pc_{\text{clear[btypes]}} \times s_{rb}. \tag{3.2}$$

The term $pc_{\text{clear[btypes]}}$ is the execution time of the permutation of buffers to be cleared. 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 target, according to Equation 3.2.

[3] s_{rb} actually represents the size of the viewport

3.4.3. Prediction Model for SwapBuffers

The SwapBuffers command indicates that the result of the previous rendering commands shall become visible. To the extent of our knowledge, there exist two technically different approaches how GPUs implement SwapBuffers:

1. The GPU renders directly to the render target: SwapBuffers only flushes the GPU pipeline and—if used—notifies the window manager. Example: nVidia Quadro 500.

2. The GPU renders to a proprietary target buffer, which often includes caches: SwapBuffers reads, converts, and copies the data of the proprietary target buffer to the render target. This is called a buffer resolve. Example: Vivante GC2000.

Obviously, the execution time of the first approach is almost negligible, while the second involves reading and writing a significant amount of data. Our model

$$m_{\text{swapbuffers}}(s_{\text{rb}}) = pc_{swapbuffers} * s_{rb} \tag{3.3}$$

depends on the copying speed per pixel data $pc_{swapbuffers}$ and the size of the render buffer s_{rb} in pixels. For the first approach, $pc_{swapbuffers} = 0$. Moreover, we apply this prediction model also if Draw commands are executed while the render target is set to a texture. In this case, the GPU driver implicitly performs a buffer resolve from the rendered content to the texture (i.e., the render target) when the render target is switched. Additionally, some GPU drivers also perform a buffer resolve to convert uniforms such as the model view projection matrix. To this end, the model $m_{\text{swapbuffers}}(s_{\text{rb}})$ is used for all buffer resolve GPU commands. Since buffer resolves that realize SwapBuffers are the typical use-case, our model was labeled SwapBuffers.

3.5. Prediction Models for Draw

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 (i.e., the OpenGL rendering pipeline), as depicted in Fig. 3.8. The execution time heavily depends on the selected OpenGL programs. The

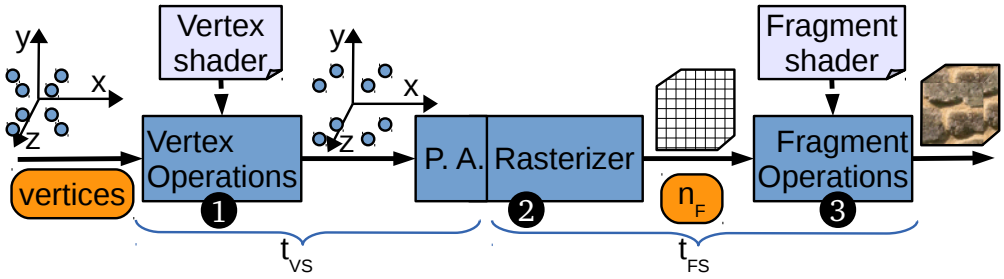


Figure 3.8.: OpenGL ES 2.0 rendering pipeline (concise)

OpenGL rendering pipeline takes as input data the vertex attributes for a set of n_v vertices, where n_v is a parameter of the OpenGL ES 2.0 Draw calls. At the first stage (❶ in Fig. 3.8), the vertex shader of the active program is executed for each vertex. 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, or rotate the vertex and create the clip coordinates [GPSM14, Ope11]. In the second stage (❷), the vertices are processed by the Primitive Assembly (P. A.), which prepares for rasterization. Then, the Rasterizer calculates the pixels (i.e., fragments) of the render buffer 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 fragment shader is executed in the third stage (❸) to assign colors to pixels, for instance, using textures. Additionally, the per-fragment operations step applies a couple of post-processing steps, such as depth test or blending, to the fragments. Finally, the output is used to update the render buffer. Many OpenGL applications use a sequence of multiple Draw commands to render a scene. This implies that CGs can contain more than one Draw command. In order to keep the prediction overhead low, the model allows to predict any number of Draw commands in one step.

To this end, the number of Draw commands is provided as n_{Calls} and multiplied by the constant overhead per Draw call ($pc_{\text{drawconst}}$). This improves the performance of the prediction since a sequence of multiple Draw commands can be predicted by a single model pass. We additionally introduce the

submodel $m_{\mathrm{VP}}(ctx)$, which estimates the execution time per vertex for the Vertex-dependent part of the OpenGL rendering pipeline, t_{VS}. Likewise, we introduce the submodel $m_{\mathrm{FP}}(ctx)$ to estimate $t_{fragmentshader(FS)}$. Both submodels depend on the current OpenGL ES 2.0 Context ctx, which contains, e.g., the fragment shader fs, the vertex shader vs, the vertex input data, and other parameters of the OpenGL rendering pipeline. In order to predict the execution time of the DRAW command, we introduce the model depicted in Equation 3.4.

$$\begin{aligned} m_{\mathrm{draw}}(n_{Calls},\, vertices,\, ctx) = {} & n_{Calls} \times pc_{\mathrm{drawconst}} \\ & + vertices.count \times m_{\mathrm{VP}}(ctx) \\ & + m_{\mathrm{nF}}(vertices,\, ctx) \times m_{\mathrm{FP}}(ctx) \end{aligned} \tag{3.4}$$

As depicted in Fig. 3.8, we assume that the execution time of the Primitive Assembly (P. A.) linearly depends on the number of vertices and therefore is considered as part of t_{VS} and $m_{\mathrm{VP}}(ctx)$. This is justified, since P. A. is performed on primitives such as triangles whose number depends on the number of vertices. Additionally, we assume that rasterization and the per-fragment operations depend linearly on the number of fragments, which is provided by the model $m_{\mathrm{nF}}(vertices,\, ctx)$. Therefore, our model does not consider them as explicit terms but as part of the execution times represented by $m_{\mathrm{VP}}(ctx)$ and $m_{\mathrm{FP}}(ctx)$. These assumptions provide a reasonable abstraction of the execution time. Typically, OpenGL ES 2.0 applications implement their main rendering functionality using sophisticated shaders. Thus, the impact of the post-processing steps is usually comparatively small, cf. Sec. 3.8.

In order to evaluate the execution time model of the DRAW command, we need the submodel $m_{\mathrm{VP}}(ctx)$ and $m_{\mathrm{FP}}(ctx)$ for the vertex processing and fragment processing, as well as the input parameters n_{Calls}, $vertices$, and m_{nF}. The parameters n_{Calls} and $vertices$ are trivial since they are directly given by the application's OpenGL API calls.

In contrast, the number of input fragments created by the rasterization step is not available a priori when the DRAW function is called. To obtain the exact number of fragments, we could emulate the vertex shader and the rasterization on the CPU. However, this would introduce high CPU overhead since the CPU would thus basically replace the GPU. To this end, we propose two heuristics for m_{nF} to estimate the number of fragments with low computational overhead. They are described in Sec. 3.5.1. The basic idea is, to approximate the size of a

triangle instead of emulating rasterization and additionally to reduce the number of vertex shader instances emulated on the CPU.

Furthermore, the model $m_{\text{draw}}(n_{Calls}, \textit{vertices}, \textit{ctx})$ requires submodels for the vertex processing $m_{\text{VP}}(\textit{ctx})$ and the fragment processing $m_{\text{FP}}(\textit{ctx})$, i.e., submodels for the execution times of vertex shader and fragment shader. To this end, we propose two kinds of submodels. The first uses the GPU Profiler for profiling the shaders online when the Predictor sees them for the first time. Unfortunately, profiling online inherently suffers from the fact that dispatching profiling CGs could impact concurrently running 3D applications. As motivated in Sec. 1.4.4, the GPU Scheduler uses the predicted execution time to ensure that low priority (non-safety critical) CGs do not prevent the timely execution of high priority (safety critical) jobs. Since a program's performance parameters are mandatory for prediction, they would be needed before they were actually determined—a chicken-and-egg problem. The execution time of profiling CGs is unknown and if the GPU Scheduler dispatches them, this could result in missed deadlines of high-priority applications. This effectively limits the use of profiling of the parameters depending on the OpenGL ES 2.0 Context to cases where no new applications join during runtime and therefore profiling can safely be performed for all applications at deploy time or at system startup. We propose a second kind of submodel, which does not require to profile each shader, even for applications which are unknown at the time of system startup and join later (such as applications newly downloaded from a third-party app store). We use machine learning in order to train submodels that provide the execution time of the shaders depending on the native GPU instruction set. The two kinds of submodels for shader prediction are presented in Sec. 3.5.2 and Sec. 3.5.3, respectively.

For some rendered scenes, the predicted execution time might lean to either overestimation or underestimation. As an optional improvement, we propose online adaption, which detects and corrects that, cf., Sec. 3.6.

3.5.1. Fragment estimation heuristics

In this section, we propose two heuristics to estimate the number of fragments by the model m_{nF}. When an application submits a sequence of DRAW calls for a frame, each CG typically contains multiple DRAW calls. This implies that the fragment estimation heuristic should support multiple DRAW calls per CG to keep the prediction overhead low. However, the number of fragments is influenced by the vertex shader setup, namely the used vertex shader program and the uniform variables that influence the vertex coordinates. Therefore, if the application changes the vertex shader setup, the current batch of DRAW commands is predicted and a new batch starts.

3.5.1.1. Approximating triangle size

The size of a triangle in a 2D coordinate system can be calculated with Equation 3.5 (see [FT09], page 34), using the x and y coordinates of the triangle's vertices a, b, and c.

$$\begin{aligned} area_{\mathrm{t}}(a,b,c) &= \frac{1}{2}\begin{vmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{vmatrix} \\ &= \frac{a.x \times (b.y - c.y) + b.x \times (c.y - a.y) + c.x \times (a.y - b.y)}{2} \end{aligned} \tag{3.5}$$

For the rare case when a triangle intersects with the border of the render buffer, we clip it to the render buffer boundaries and use the intersection points to create a new convex polygon. The area is then calculated by summing up the areas of the constituent triangles of the convex polygon. The respective algorithm is straight forward, but lengthy and thus omitted for the sake of readability.

The principle how GPUs create fragments is specified by the OpenGL ES 2.0 standard as follows.

> "The rule for determining which fragments are produced by polygon rasterization is called point sampling. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization." ([ML10], page 57)

Rasterization is a discretization of the continuous 2D space assumed by Equation 3.5. This implies that with OpenGL ES 2.0 the number of fragments

produced by a triangle can vary depending on the triangle translation. Unfortunately, this means that the exact number of fragments of a triangle can only be determined exactly by emulating the Vertex Operations and the Rasterizer (cf., Fig. 3.8). Since this would require way too much CPU resources for the prediction, we use Eq. 3.5 with clipping to approximate the size of a triangle. Next, we discuss the potential inaccuracy introduced by this approximation.

Due to the discretization of the fragments, the number of fragments can be either overestimated or underestimated by our approximation. Fig. 3.9 depicts

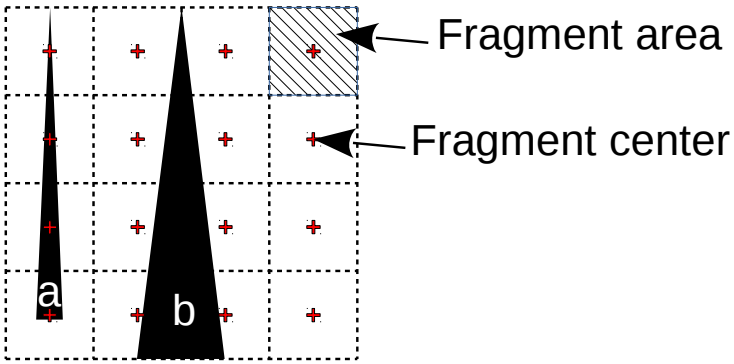


Figure 3.9.: Example of possible deviation of triangle size approximation

an example which contains a small render buffer that is 4 pixels wide and 4 pixels high. Its 16 squares represent the areas covered by the 16 fragments. The center of each fragment is marked by a cross. As quoted from the OpenGL ES 2.0 specification, a fragment is created, iff its center is covered by a triangle. In our example, triangle “a” covers four centers of fragments and thus would show up as a row of four fragments. However, the actual size of the triangle is much smaller than four, i.e., the number of fragments would be underestimated in this case. In contrast, triangle “b” covers no center of any fragment and thus would not show up at all on the render buffer. Although triangle “b” is actually larger than triangle “a”, it results in much less fragments due to the discretization of fragments. For non-malicious applications (i.e., applications that do not intentionally try to create wrong estimations), all these examples are statistically irrelevant and do not result in significant approximation errors. The cases of malicious applications that try to achieve overestimation are not critical, since they would lead to an overestimation of the execution time of a CG, which is disadvantageous to these applications, only. However, applications that try to achieve underestimation of the number of fragments could be critical. For instance, for potentially malicious third-party applications, thin

triangles could be detected and be overestimated using the longest side and all size values could be rounded up.

3.5.1.2. Emulating the vertex coordinates calculation

In the vertex shader code, the vertex coordinates are assigned to the special variable "gl_Position". In order to determine the size of a triangle, the three coordinates of "gl_Position" must be known. To this end, we emulate the vertex coordinate calculation on the CPU and then calculate the window coordinates. The window coordinates represent the pixel coordinates on the render target. Fortunately, the calculations of varying variables—which is often much more complex—must not be emulated. In order to obtain window coordinates from the vertex input data, three steps are required:

1. Emulate the calculation of the "gl_Position" of the vertex shader code to obtain the vertex coordinates[4] x, y, z, and w.

2. Calculate the normalized device coordinates x_d, y_d, and z_d [MGS09], i.e.,
$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}.$$

3. Calculate window coordinates using the viewport parameters (as given by the "glViewport" API call) [GPSM14]:
$$\begin{pmatrix} x_w \\ y_w \end{pmatrix} = \begin{pmatrix} \frac{w_{viewport}}{2} \times x_d + x_{viewport} + \frac{w_{viewport}}{2} \\ \frac{h_{viewport}}{2} \times y_d + y_{viewport} + \frac{h_{viewport}}{2} \end{pmatrix}$$

Function *emulate⊿*: For later use, we define the function *emulate⊿*, which calculates the area of the projected triangle based on a triangle's input vertex data. The function first calculates the window coordinates from the vertex input data for each of the triangle's three vertices. As described in Sec. 3.5.1.1, the window coordinates (which depend on the vertex shader) are used to estimate the size of the triangle. In this calculation, the OpenGL state for culling ("GL_CULL_FACE") and facing ("GL_FRONT_FACE") are considered properly. The calculated triangle area is returned by *emulate⊿*. If the calculated area is negative, the triangle is not facing towards the camera and will not produce any fragment; in this case *emulate⊿* returns 0.

[4]The values z and w are optional and default to 0 and 1, respectively.

Listing 3.2: Record trian1gle samples data after DRAW calls

```
1 function record_triangle(triangle T, int vecSize)
2     recordedTriangles++
3     IF (random_int() MOD p^{-1}) == 0)
4         IF (samplesInBuffer >= BUF_SIZE)
5             IF (p == p_min):
6                 TSarea += emulate⊿(unify(T, vecSize), vertex
  shader)
7                 numTS++
8                 return
9             FOR i=2 TO BUF_SIZE - 1 STEP 2:
10                TSbuf[i/2] = TSbuf[i]
11            p = p/2;
12            samplesInBuffer = (BUF_SIZE / 2)
13        TSbuf[samplesInBuffer] = unify(T, vecSize)
14        samplesInBuffer++
```

Next, we present two heuristics that estimate the number of fragments using window coordinates and the triangle size approximation presented in Sec. 3.5.1.1.

3.5.1.3. Triangle samples heuristic

The triangle samples heuristic approximates the number of fragments by using a typically small, randomly selected subset of the rendered triangles. For these randomly selected samples we emulate the vertex coordinate calculation (cf. Sec. 3.5.1.2) and estimate the size (cf. Sec. 3.5.1.1). From the DRAW calls we know the total number of rendered triangles, which is used to extrapolate the area covered by the selected triangle samples. The size of the subset is quite important for good prediction accuracy, since a small subset typically implies low accuracy. For small models, the additional challenge is, to use a subset that contains a sufficiently high number of triangles (must obviously be greater than 0). We therefore propose to use a triangle buffer of a fixed size and a varying probability at which triangle samples are selected.

Next, we describe the approach in more detail. If the triangle samples mode is active, after a DRAW call has been forwarded to the native driver the function "record_triangle" is called for each rendered triangle. The "record_triangle" function, depicted in Listing 3.2, is called for each triangle that has been passed to the OpenGL API using a DRAW call. The number of rendered triangles is counted in "recordedTriangles" (Line 2). A triangle is selected as a sample with probability p (Line 3). For a new DRAW batch, initially $p = 1$, which allows to

Listing 3.3: Using triangle samples to predict the number of fragments

```
1 function trace_triangles_buf(vertex shader)
2     FOR i=0 TO samplesInBuffer:
3         TSarea += emulate⊿(TSbuf[i], vertex shader)
4         numTS++
5     estimatedArea = (TSarea / numTS) * recordedTriangles
6     p = 1
7     samplesInBuffer = TSarea = numTS = recordedTriangles = 0
8     return estimatedArea
```

accurately predict 3D objects with few triangles. Since p is only reduced by half, p^{-1} is always an integer of the form 2^x. The buffer used to store the triangle samples has a size limit of BUF_SIZE (256 by default). If all slots are occupied (Line 4), the probability is either reduced by half, or if the minimum probability p_{min} (Line 5, $p_{min} = 2^{-7}$ by default) is reached, further samples (beyond the buffer capacity) are selected with probability p_{min}. Selecting a sample means to estimate its area with $emulate_{\triangle}$ and to increment the samples counter numTS (Lines 6–7). If $p > p_{min}$, every second element of the buffer is removed (Lines 9–10), the probability p and the buffer's samples counter are reduced by half (Lines 11–12). A unified copy of the triangle is then kept in the buffer and the buffer's samples counter is incremented, accordingly (Lines 13–14). The parameter vecSize of the function "unify" denotes how many coordinate elements the vertex shader uses for each vertex and is between 2 and 4 (depending on whether "z" and "w" values are given). Since all later calculations expect a 4-component vector (x,y,z,w), "unify" returns a triangle consisting of three 4-component vectors. If an application uses only 2 or 3 components, the "z" or "w" attributes default to 0 and 1, respectively (cf., [ML10]).

When a DRAW batch is complete, e.g., if the CG is submitted or the vertex shader changes, the estimated number of fragments is calculated by the function "trace_triangles_buffer" depicted in Listing 3.3. For each triangle sample in the buffer, its area is calculated and the samples counter numTS is incremented (Lines 2–4). The estimated area is calculated by extrapolating the average triangle size of all samples to the number of recorded triangles (Line 5). Since now the DRAW batch is complete, the probability p is reset to 1 and all other state variables are reset to 0 (Lines 6–8).

Fig. 3.10 shows the relation between the number of triangles in a DRAW batch and the average number of triangle samples. For up to $BUF\_SIZE = 256$

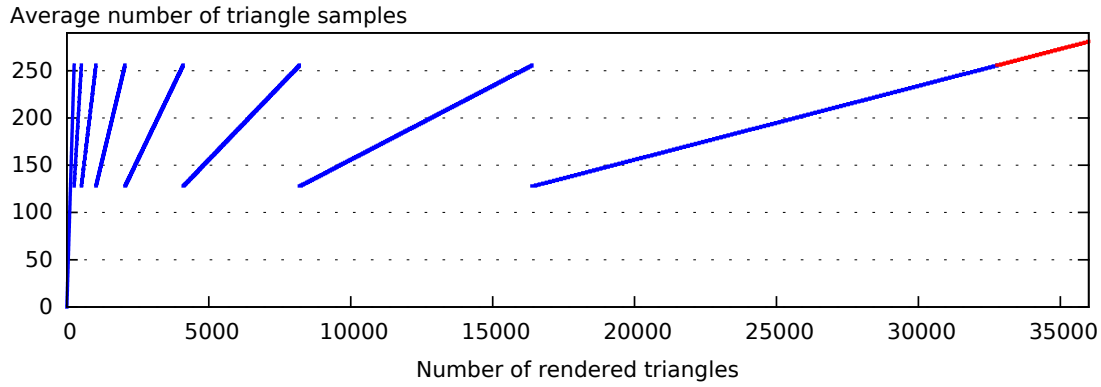


Figure 3.10.: Average triangle samples depending on number of rendered triangles

triangles $p = 1$ (each triangle is a sample). For more triangles, the probability decreases by 50 % in each step until the defined minimum $p_{min} = 2^{-7}$ is reached. Then, the line continues in red color to indicate that the buffer is full and samples beyond buffer capacity are selected with $p = p_{min}$.

The triangle samples (TS) approach tries to prevent underestimation. If an application draws a mixture of big and small triangles of ratio r, the set of samples likely contains big and small triangles of a ratio close to r.

3.5.1.4. Bounding box heuristic

An alternative heuristic for m_{nF} uses a bounding box that encloses all vertices. Instead of calculating the area of actually rendered triangles, this approach performs the geometric transformation of the vertex shader only on the eight vertices of the bounding box, as depicted in Fig. 3.11. The result is the projected bounding box. Typically, vertex shaders multiply the vertex data with one or two 4x4 matrices, which limits the changes of each vertex to translation, scaling, and rotation [MGS09]. This implies that all vertex coordinates

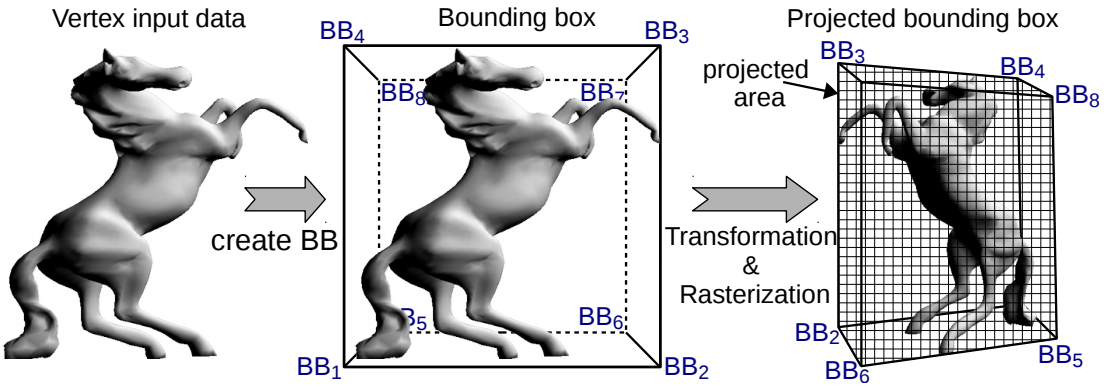


Figure 3.11.: Bounding box applied on a horse model

computed by the vertex shader are contained in the projected bounding box. In the last step, we calculate the number of pixels that cover the rasterized 2D area of the projected bounding box. Since typical models try to minimize overpainting, this number can typically be used for m_{nF}. Next, we describe in more detail how the bounding box is created and how its area is calculated.

When the application issues a DRAW call, each rendered triangle is traced. Tracing means the minimum and maximum values of the x, y, and z values are tracked. If the vertex data is a four-component vector, the x, y, and z components are normalized by dividing each of them by the w component [MGS09]. The bounding box BB is created as depicted in Equation 3.6.

$$BB = \begin{array}{c} \\ x \\ y \\ z \\ w \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \left[\begin{array}{cccccccc} x_{min} & x_{max} & x_{max} & x_{min} & x_{min} & x_{max} & x_{max} & x_{min} \\ y_{min} & y_{min} & y_{max} & y_{max} & y_{min} & y_{min} & y_{max} & y_{max} \\ z_{min} & z_{min} & z_{min} & z_{min} & z_{max} & z_{max} & z_{max} & z_{max} \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{array}\right] \end{array} \tag{3.6}$$

BB_1 represents the bottom left front vertex of the bounding box. The front rectangle is represented by the BB_1 to BB_4 in counter-clockwise order. Similarly, BB_5 to BB_8 represent the back rectangle in counter-clockwise order. The forth component w is always 1, either because it is missing in the input data, or because it was already used to normalize x, y, and z. The area covered by the bounding box is calculated by aggregating the area covered by all six faces, cf., Eq. 3.7.

$$\begin{aligned}
1:\ BBarea \quad &= emulate_{\triangle}(BB_1, BB_2, BB_3) \quad + emulate_{\triangle}(BB_1, BB_3, BB_4) \\
2: \quad &+ emulate_{\triangle}(BB_2, BB_6, BB_7) \quad + emulate_{\triangle}(BB_2, BB_7, BB_3) \\
3: \quad &+ emulate_{\triangle}(BB_6, BB_5, BB_8) \quad + emulate_{\triangle}(BB_6, BB_8, BB_7) \\
4: \quad &+ emulate_{\triangle}(BB_5, BB_1, BB_4) \quad + emulate_{\triangle}(BB_5, BB_4, BB_8) \\
5: \quad &+ emulate_{\triangle}(BB_4, BB_3, BB_7) \quad + emulate_{\triangle}(BB_4, BB_7, BB_8) \\
6: \quad &+ emulate_{\triangle}(BB_5, BB_6, BB_2) \quad + emulate_{\triangle}(BB_5, BB_2, BB_1)
\end{aligned} \tag{3.7}$$

The Lines 1 to 6 represent the front, right, back, left, top, and bottom faces, respectively. For each face, the area is calculated using the function $emulate_{\triangle}$ on its two spanning triangles. The area covered by the bounding box is an upper bound to the area covered by pixels. However, the actually covered area is often smaller. For instance, in the example in Fig. 3.11 about 30 % of the projected

bounding box area is covered by a rendered pixel. However, part of covered pixels was rendered by multiple fragments, i.e., areas where one part of the horse overlaps another and the front fragment is rendered after the hidden fragment. We reflect this in our prediction model m_{nF} by a coverage factor that is multiplied with the number of pixels covered by the bounding box to obtain the estimated number of fragments.

The bounding box (BB) approach tries to provide a good average accuracy. To this end, m_{nF} is determined such that for different kinds of applications, the average error is small.

The use of the BB approach is restricted to vertex shaders where the calculation of "gl_Position" is equivalent to multiplications of the vertex input data by model matrix, view matrix, or projection matrix. While this is the typical case for almost all existing shader programs, there exist exceptions (e.g., the "jellyfish" scene of glmark2). In such cases, the BB approach should not be used[5], since pixels could be rendered outside the area covered by the bounding box.

[5]Our implementation would detect that and prevents using the BB approach by mistake.

3.5.2. Shader model: based on profiling

In this section, we describe how profiling is used for the prediction of the vertex processing time and the fragment processing time, represented by the models $m_{\mathrm{VP}}(ctx)$ and $m_{\mathrm{FP}}(ctx)$, respectively.

3.5.2.1. Prediction model

For the profiling-based shader prediction submodels, the following simple vertex shader model is used.

$$m_{\mathrm{VP}}(ctx) = pv_{\mathrm{vertex\_shader}} \tag{3.8}$$

In Equation 3.9, the fragment shader model is depicted.

$$m_{\mathrm{FP}}(ctx) = pv_{\mathrm{fragment\_shader}} + \begin{cases} pc_{\mathrm{depth}}, & \text{if } ctx.dt \\ 0, & \text{otherwise} \end{cases} + \begin{cases} pc_{\mathrm{blending}}, & \text{if } ctx.blending \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

It uses the shader execution times $pv_{\mathrm{vertex\_shader}}$ and $pv_{\mathrm{fragment\_shader}}$ determined by the GPU Profiler and the overhead introduced by depth test and blending. Both, depth test and blending, are only considered if they are enabled by the 3D application for the current OpenGL ES 2.0 Context ctx, which is determined by the OpenGL ES Context Monitor. We did not include Blending in this submodel, since in our evaluations we observed that Blending does not depend linearly on the number of fragments and thus requires a more expressive model (such as our MARS model in Sec. 3.5.3.3).

3.5.2.2. Profiling of shader execution time

In this section, we describe how the GPU Profiler determines $pv_{\mathrm{vertex\_shader}}$ and $pv_{\mathrm{fragment\_shader}}$. Both parameters are OpenGL program-dependent and therefore must be determined for each OpenGL program before its CGs can be predicted. When a OpenGL program is seen for the first time, the GPU Profiler executes it in an instrumented profiling mode. The profiling function performs the following major steps to measure the shader performance.

1. Query the program's attributes and uniforms since data must be provided for them.
   Used OpenGL command: glGetProgramiv.

2. Activate vertex shader profiling, configure the OpenGL context dedicated for profiling.

Used OpenGL commands: eglMakeCurrent, glUseProgram, glEnable, and glDepthFunc.

3. Generate VBOs for vertex shader profiling and fragment shader profiling. Used OpenGL command: glGenBuffers.

4. For each vertex attribute, the data type and the number of components are obtained and used to create vertex attribute data.
   Used OpenGL commands: glGetActiveAttrib, glGetAttribLocation, glBindBuffer, and glBufferData.
   a) For vertex shader profiling, the data is a sequence of a high number of minimum-sized triangles (to produce 0 fragments).
   b) For fragment shader profiling, the data is a sequence of a small number of huge triangles, each covering half of the viewport.

5. For each uniform variable, the data type and number of components are obtained and used to create uniform variable data.
   Used OpenGL commands: glGetActiveUniform and glGetUniformLocation.
   a) For uniform variables of type "GL_SAMPLER_2D", a dedicated texture based on dummy data using constant width, height, data format, and data type is used.
      Used OpenGL commands: glGetUniformLocation, glBindTexture, glTexImage2D, glActiveTexture, and glUniform1i.
   b) Uniform variables of type "GL_SAMPLER_CUBE" are handled the same way as "GL_SAMPLER_2D", except that a single dedicated texture is used for all six sides of the cube.
      Used OpenGL commands: glGenTextures, glBindTexture, glTexImage2D, glActiveTexture, and glUniform1i.
   c) For all other data types, the required amount of data—depending on type and number of vector elements—is used for the respective "glUniform*" call.
      Used OpenGL commands: glUniform*f, glUniform*fv, glUniform*i, glUniform*iv, and glUniformMatrix*fv (a "*" represents the numbers 1 through 4).

6. The vertex shader profiling issues a defined number of Draw commands, each followed by a "glFlush()", which emits a CG. Used OpenGL

commands: glBindBuffer, glVertexAttribPointer, glEnableVertexAttribArray, glDrawArrays, and glFlush.

7. Activate fragment shader profiling.
   Used OpenGL commands: glDisableVertexAttribArray.

8. The fragment shader profiling issues a defined number of DRAW commands, each followed by a "glFlush()".
   Used OpenGL commands: glBindBuffer, glVertexAttribPointer, glEnableVertexAttribArray, glDrawArrays, and glFlush.

9. Free all allocated resources and switch back to the native context of the application.
   Used OpenGL commands: glDisableVertexAttribArray, glDeleteBuffers, glDeleteTextures, and eglMakeCurrent.

After a CG has finished execution, the Execution Time Monitor provides its measured execution time and the number of emitted CGs. To calculate $pv_{\text{vertex\_shader}}$ and $pv_{\text{fragment\_shader}}$, the terms in Table 3.3 are used.

Table 3.3.: Nomenclature of shader profiling calculations

Term	**Description**
t_{vs}	Total measured execution time of all CGs of the vertex shader profiling
t_{fs}	Total measured execution time of all CGs of the fragment shader profiling
$\#VS\_CGs$	Number of CGs emitted during vertex shader profiling
$\#FS\_CGs$	Number of CGs emitted during fragment shader profiling
$\#VS\_DRAWS$	Number of Draw commands (i.e., profiling samples) emitted during vertex shader profiling
$\#FS\_DRAWS$	Number of Draw commands (i.e., profiling samples) emitted during fragment shader profiling
$\#VS\_TRIANGLES$	Number of triangles rendered at vertex shader profiling
$\#FS\_TRIANGLES$	Number of triangles rendered at fragment shader profiling
s_{rb}	Size of the render buffer in pixels

The measured execution time t_{vs} of the vertex shader profiling is used to calculate

$$pv_{\text{vertex\_shader}} = \frac{t_{vs} - \#VS\_CGs \times m_{\text{flush}}() - \#VS\_DRAWS \times pc_{\text{drawconst}}}{\#VS\_TRIANGLES * 3}. \tag{3.10}$$

The term #VS_DRAWS represents the number of Draw commands. Each Draw command draws #VS_TRIANGLES triangles, each consisting of 3 vertices. The number of CGs is represented by num_{CGs}, since the GPU driver might emit more than one CG per Draw call, e.g., if the command buffer needs to be switched.

The measured execution time t_{fs} of the fragment shader profiling is used to calculate

$$pv_{\text{fragment\_shader}} = \frac{t_{fs} - \#FS\_TRIANGLES \times 3 \times pv_{\text{vertex\_shader}} - fsub}{\#FS\_TRIANGLES * \frac{\mathtt{s}_{rb}}{2}}. \tag{3.11}$$

where

$$fsub = \#FS\_CGs \times m_{\text{flush}}() - \#FS\_DRAWS \times pc_{\text{drawconst}}$$

Since OpenGL does not allow to run fragment processing separate from vertex processing, the time t_{fs} includes also a small amount of vertex processing time. The determined parameter $pv_{\text{vertex\_shader}}$, which estimates the execution time per vertex, and the number of vertices (using #FS_TRIANGLES) is used to calculate this vertex processing time. The overhead to flush a CGs and the number of Draw commands is again also considered.

3.5.3. Shader model: based on machine learning

For the prediction of Draw commands, obtaining the performance parameters that depend on the OpenGL ES 2.0 Context is mandatory. In Section 3.5.2.2 we describe how profiling during runtime can be used to determine them. An implementation based on profiling during runtime can be easily used for different GPUs and platforms since often none or only minor changes are required. However, it suffers from the fact that dispatching profiling CGs could violate a deadline of an application that has higher priority. For cases where new applications can join during runtime of important 3D applications, the proposed profiling during runtime cannot be used safely. In this section, we therefore describe a machine-learning-based approach to determine the performance parameters that depend on the OpenGL ES 2.0 Context without profiling. Our models determine the execution time per vertex $m_{\mathrm{VP}}(ctx)$ and per fragment $m_{\mathrm{FP}}(ctx)$ (cf., Sec. 3.5). These models are tailored for the specific GPU model and platform, including determining and extracting the relevant features. However, this one-time effort is often justified by the achieved benefits, since for an automotive scenario in a given vehicle model the used GPU and platform are changed rarely.

We thoroughly analyzed the GPU behavior to determine suitable features for a machine learning algorithm. This analysis needs to be done for each GPU platform, since GPU behavior might differ between models or vendors. The model presented in this section is based on the analysis of the Vivante GC2000 3D GPU integrated into a Freescale i.MX6 SABRE Automotive platform. However, the general approach can also be used to create an adapted model for further GPU platforms.

3.5.3.1. Introduction and Overview

Choosing the right machine learning concept for a problem is crucial to get reasonable results. In Section 3.1.3, we introduced the relevant machine learning concepts. The desired output is the estimated execution time. Thus, supervised learning is preferred over unsupervised learning. For supervised learning, many concepts exist (cf., Sec. 3.1.3). We analyzed, whether a linear regression model can be used for execution time prediction. In Fig. 3.12, we depict the execution time of a simple vertex shader that does not produce any fragments and uses a varying number of "ADD" commands. Additionally, we used between 1 and 4 vertex attributes of type vec4. Each curve starts with a range where a more

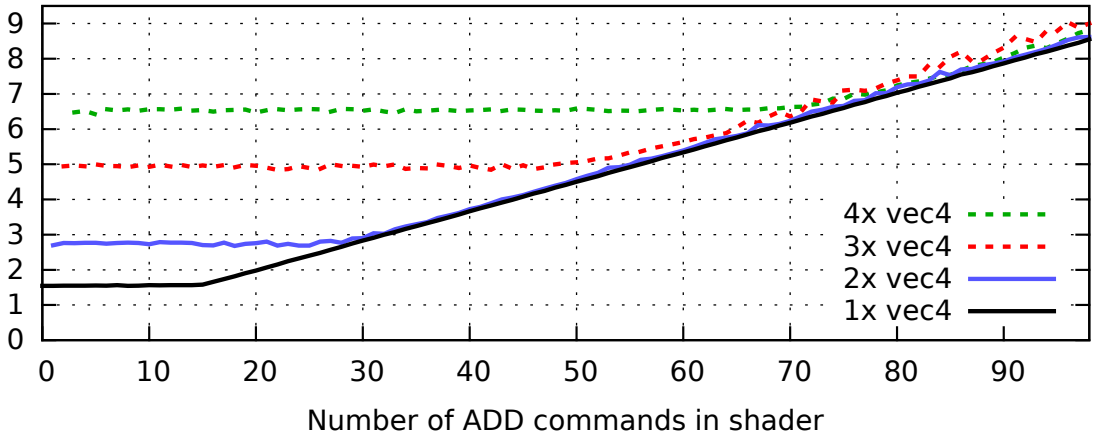


Figure 3.12.: Execution time of VS depending on the number of vertex attributes

complex computation (i.e., more ADD commands) does not increase execution time; after that it continues linearly. The constant execution time, as well as the point of transition to linear execution time, are unfortunately depending on the vertex attributes. In more detail, the following behavior of the Vivante GC2000 GPU was observed and needs to be considered at model creation:

- The set of vertex attributes in the vertex shader defines a minimum vertex shader execution time (Fig. 3.12).
- The set of varying variables influences the speed of GPU instructions in the vertex shader.
- The set of varying variables and fragment post-processing parameters define a minimum fragment shader execution time.
- To a minor degree, the set of varying variables influences the execution speed of fragment shader instructions.
- Accessing textures involves memory access, which depends on memory latency and interferes with other memory-related executions of the shaders.

Although linear regression can support piece-wise linear functions, the transition points must be constant. Since Fig. 3.12 shows that the transition point of vertex shaders depends on the input data, a single linear model is insufficient. To cope with this situation, a combination of two linear models could be used where one predicts the minimum execution time and the other the part that depends on the used GPU instructions. However, the latter

depends also on the set of varying variables. Therefore, for each possible combination of vertex attributes a feature would have to be created. Let $numVars_{MAX}$ be the maximum supported number of varying variables. For each of the 97 different GPU instructions and up to $numVars_{MAX}$ varying variables, $\sum_{i=0}^{numVars_{MAX}} \binom{4+i-1}{i}$ features would be required to predict the part of the vertex shader execution time that depends on the used GPU instructions. E.g., for $numVars_{MAX} = 5$ (supporting up to 5 varying variables), 12222 features would be required. This is quite much, since for each prediction a feature vector of this size would have to be created and evaluated on the embedded system—a quite compute-intensive undertaking. Additionally, this approach suffers from the fact that a maximum number of varying variables $numVars_{MAX}$ must be specified. If the number of varying variables should be higher than $numVars_{MAX}$, no knowledge would be available. Therefore, linear regression is not expressive enough for this GPU model. Solutions with sufficient expressiveness are non-linear models such as multivariate adaptive regression splines (MARS) [Fri91] or continuous piecewise-linear neural networks (CPLNN) such as [WHJ08].

When choosing a machine learning algorithm, we considered—besides expressiveness—the expected CPU time for prediction

Using multi-layered CPLNNs not only requires much more training data but also ends up in a much bigger model with many neurons and many input parameters. While the training can be performed on high-performance compute clusters, the prediction must be performed on the embedded platform, which has constrained resources. For cost reasons, reducing resource consumption is a major goal of the automotive domain. For this reason, CPLNNs and deep learning were not considered.

We decided to use MARS [Fri91] models for the following reasons. When a MARS model is created, all terms of low significance are removed and therefore do not cause computational effort at prediction. Additionally, MARS provides just the right amount of expressiveness for the GPU behavior observed. Note, however, that for other GPU models, other machine learning concepts might suite better. To create our MARS models, we used the "earth" implementation in R [Mil11]. R is a widely-used software environment for statistical computing and evaluations [RPr17, MB10]. We always used the option "fast.k=0", which disables a heuristic that speeds up model creation but often failed to identify important terms. For a similar reason, we used "nk=1000" to use a large search space. The execution time of a shader instance depends on the time consumed

by a GPU shader core and auxiliary time. The time consumed by a GPU shader core linearly increases with the number of GPU commands, for which we introduce the submodels $m_{\text{cmdsVS}}(ctx)$ and $m_{\text{cmdsFS}}(ctx)$. If a shader contains texture lookup commands such as "texture2D", this introduces memory access overhead, for which we introduce the submodel $m_{\text{texld}}(shader)$. The auxiliary time represents the effort of loading the respective input data (vertex attributes or varying variables) and performing Primitive Assembly or fragment post-processing. For this purpose, we introduce the submodels $m_{\text{auxVS}}(ctx)$ and $m_{\text{auxFS}}(ctx)$. Our measurements show that the GPU performs this effort in parallel to the execution performed by the GPU shader cores and thus constitutes the minimum execution time. If the auxiliary time is higher than the GPU shader core time, the utilization of the shader cores is reduced. Similarly, if the auxiliary time is lower, the respective hardware subsystems are repeatedly idle. In order to keep the model complexity—and thus the computational effort of prediction—small, for both, vertex and fragment shader, the following combined models are therefore used to estimate the draw execution time $m_{\text{draw}}(n_{Calls}, vertices, ctx)$ (cf., Sec. 3.5):

$$m_{\text{VP}}(ctx) = max(m_{\text{auxVS}}(ctx), m_{\text{cmdsVS}}(ctx) + m_{\text{texld}}(ctx.vs))$$
$$m_{\text{FP}}(ctx) = max(m_{\text{auxFS}}(ctx), m_{\text{cmdsFS}}(ctx) + m_{\text{texld}}(ctx.fs))$$

To obtain good models, a huge number of different shaders is required. Since the set of available existing 3D applications and their shaders is orders of magnitudes too small, we created tailored training data for supervised learning by varying the parameters used for features. The output vector always has only one element: the relevant part of the measured execution time. Parts of the execution time that were not relevant, were subtracted from the actually measured execution times, namely pc_{flush}, $n_{Calls} \times pc_{\text{drawconst}}$, and the unavoidable vertex shader execution times of the fragment shader training data.

Nomenclature The submodels use the terms listed in Table 3.4. These terms

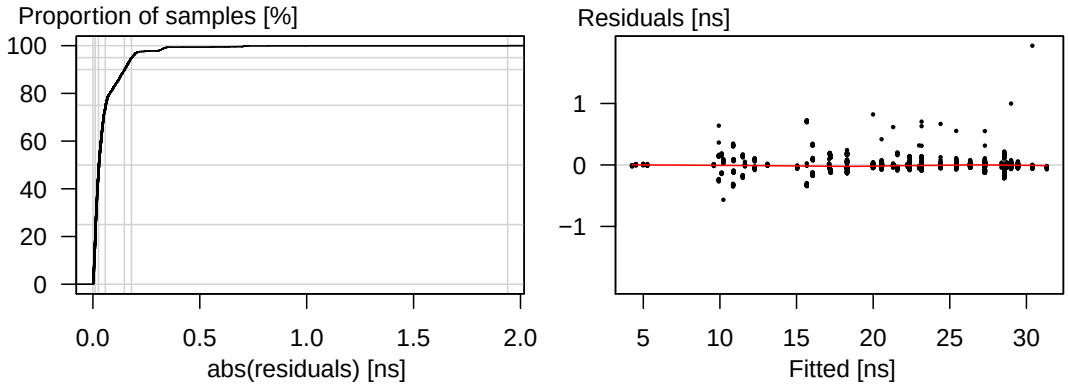
Table 3.4.: Nomenclature of MARS submodel terms

Term	Description
$\#A_1$, $\#A_2$, $\#A_3$, $\#A_4$	Number of vertex attributes of type float, vec2, vec3, vec4
numAttrs	$\#A_4 + \#A_3 + \#A_2 + \#A_1$ (number of vertex attributes)
numAttrFloats	$4\#A_4 + 3\#A_3 + 2\#A_2 + \#A_1$ (vertex attribute components)
$\#V_1$, $\#V_2$, $\#V_3$, $\#V_4$	Number of varying variables of type float, vec2, vec3, vec4
numVars	$\#V_4 + \#V_3 + \#V_2 + \#V_1$ (number of varying variables)
numVarFloats	$4\#V_4 + 3\#V_3 + 2\#V_2 + \#V_1$ (varying variable components)
$\#VS_{cmds} \in \mathbb{N}^{97}$	GPU instructions per vertex shader instance
$\#FS_{cmds} \in \mathbb{N}^{97}$	GPU instructions per fragment shader instance

were carefully selected based on the observed GPU behavior on large training data, as suggested by others (e.g., [GE03]). Next, we present our five MARS submodels.

3.5.3.2. Predicted Auxiliary Vertex Shader Time

As discussed earlier, the auxiliary vertex shader time depends on the set of used vertex attributes. For few commands, the execution time does not depend on the number of commands. For more commands, it moves on to a linear dependency. Next, we explain the respective submodel $m_{\text{auxVS}}(ctx)$. We analyzed how the permutation of vertex attributes influences the execution time and created training data using all possible permutations of one to four vertex attributes (typical applications use between one and three vertex attributes). The order in which the vertex attributes were defined had no impact. Therefore, we used $\#A_1$, $\#A_2$, $\#A_3$, and $\#A_4$ as features. Additionally, we used *numAttrs* and numAttrFloats, which allows MARS to build a model with less terms without compromising accuracy. We created the submodel $m_{\text{auxVS}}(ctx)$ with $degree = 6$ and $thresh = 1e-08$ (cf., Sec. 3.1.3.2). With 34 terms, a very good prediction was achieved. The cumulative distribution—depicted in Fig. 3.13a—shows that 95 % of the values have an error of less than 0.2 ns per shader instance and the worst error observed was below 2 ns. The estimations were in the range of about 4 ns to 33 ns. The residuals vs. fitted plot (Fig. 3.13b) shows that for the vast majority, underestimation is of similar order than overestimation.

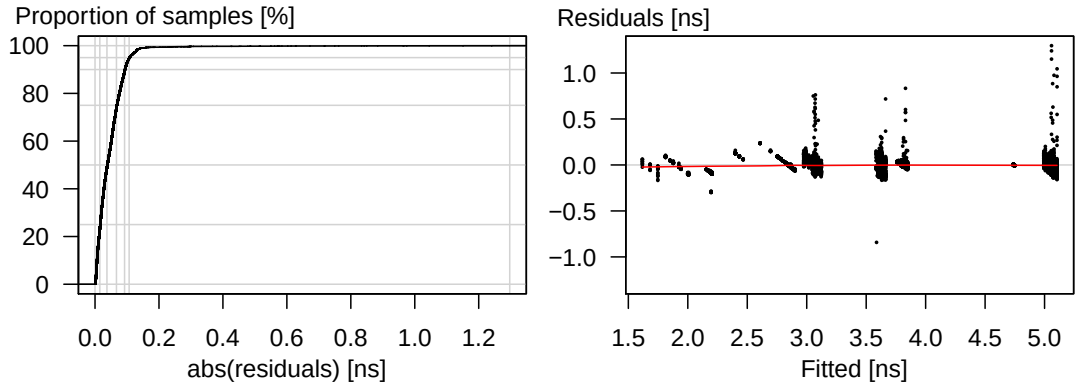


(a) Cumulative distribution (b) Residuals vs. fitted

Figure 3.13.: Error of submodel for auxiliary vertex shader execution time

3.5.3.3. Predicted Auxiliary Fragment Shader Time

For a fragment shader, the varying variables serve as input, comparable to what vertex attributes are for the vertex shader. The auxiliary fragment shader time depends on the set of used varying variables. Therefore, the submodel of the auxiliary fragment shader time $m_{\text{auxFS}}(ctx)$ is similar to the submodel of the auxiliary vertex shader time $m_{\text{auxVS}}(ctx)$. Next, we explain the respective model $m_{\text{auxFS}}(ctx)$. The auxiliary fragment shader time depends on the used varying variables, only. We therefore analyzed how the permutation of varying variables influences the execution time and created training data using all possible permutations of one to four varying variables. The order in which the varying variables were defined had no impact. Therefore, we used $\#V_1$, $\#V_2$, $\#V_3$, and $\#V_4$ as features. Additionally, we used *numAttrs* and numAttrFloats, which allow MARS to build a model with less terms without compromising accuracy. Moreover, blending and depth test were used as binary features, i.e., a 0 means "disabled", a 1 means "enabled". For $m_{\text{auxFS}}(ctx)$, 38 terms were selected. The features representing Depth test and Blending were determined as the most important. This indicates that they introduce overhead due to additional memory operations. The cumulative distribution of the absolute error is given in Fig. 3.14a and shows that 95 % of the samples are fitted with an absolute error of less than only about 0.1 ns. Given that the fitted values go up to about 5.12 ns (cf., Fig. 3.14b), the achieved accuracy is good.



(a) Cumulative distribution

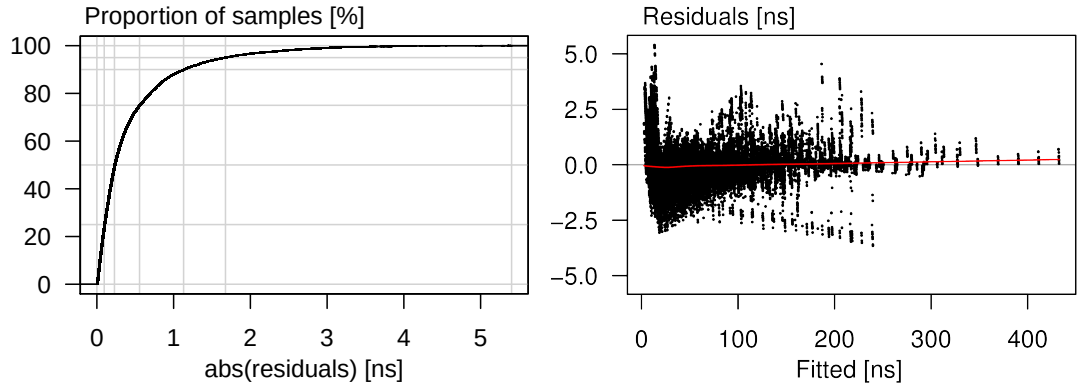
(b) Residuals vs. fitted

Figure 3.14.: Error of submodel for auxiliary fragment shader execution time

3.5.3.4. Predicted Vertex Shader Command Time

Next, we describe how the submodel for the vertex shader commands $m_{\text{cmdsVS}}(ctx)$ was created. We measured all permutations of one to four varying variables and 1 to 20 repetitions of 33 OpenGL Shading Language expressions creating different GPU commands. We ignored all CGs that did not clearly exceed the auxiliary vertex shader time $m_{\text{auxVS}}(ctx)$. The execution time of a vertex shader that exceeds the auxiliary vertex shader time depends on the number of the respective GPU commands and the set of varying variables. It is independent from the used vertex attributes, though.

Therefore, we used $\#V_1$, $\#V_2$, $\#V_3$, $\#V_4$, and the number of occurrences of all available GPU commands $\#VS_{cmds}$ as features. Additionally, using the features *numVars* and *numVarFloats* allowed MARS to achieve good accuracy using less terms. As an exception, we did not include the "MOV" command in the set of features. The "MOV" command copies data and is—according to our observations—often executed in parallel to other command types. For instance, a sequence of "MOV"/"ADD" command pairs takes about as long to execute as the same number of "ADD" commands without interleaving "MOV" commands. The submodel was created with $degree = 4$ and $thresh = 0.0000005$. Additionally, the set of all features representing numbers of GPU commands was passed in the parameter *linpreds* to indicate that these terms shall enter only linearly into the submodel. Limiting the number of occurrences of GPU commands to linear dependencies is indicated by the data and significantly speeds up model creation. A total of 465 terms were selected. Each term contains no more than one feature representing the number of a GPU command. Each of those GPU command



(a) Cumulative distribution (b) Residuals vs. fitted

Figure 3.15.: Error of submodel for vertex shader commands execution time

features was used in up to 20 terms: One that depends only on the number of command occurrences, the others depending on *numVars* and *numVarFloats*. For about 95 % of the cases, the deviation from the submodel is normally distributed with an error of less than 1.7 ns (cf., Fig. 3.15a). Since the GPU behavior is quite complex, the achieved maximum error of less than 5.4 ns is very good. The residuals vs. fitted plot depicted in Fig. 3.15b shows that this small error was achieved notwithstanding the residuals reach up to about 430 ns.

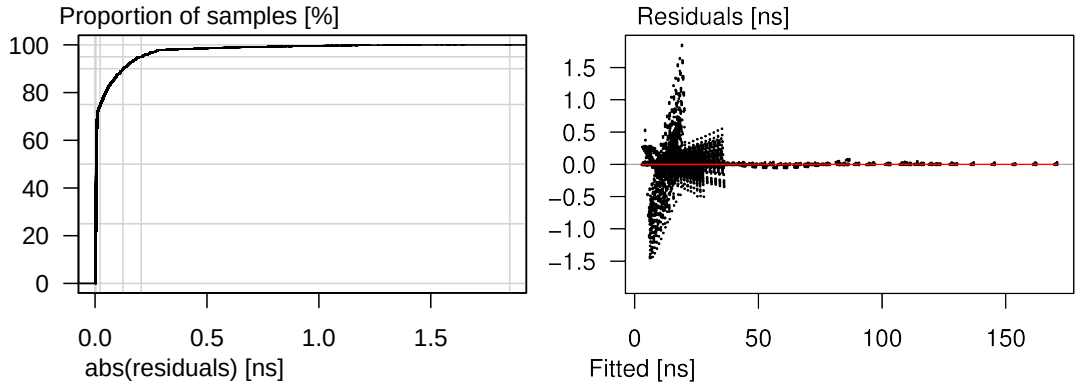
3.5.3.5. Predicted Fragment Shader Command time

In this section we describe how the submodel for the fragment shader commands $m_{\text{cmdsFS}}(ctx)$ was created. We measured all permutation of one to four varying variables and 1 to 20 repetitions of 34 OpenGL Shading Language expressions creating different GPU commands. We ignored all CGs that did not clearly exceed the auxiliary fragment shader time $m_{\text{auxFS}}(ctx)$.

The fragment shader command time depends mainly on the number of executed GPU instructions. Therefore, we used the vector $\#FS_{cmds}$ as feature, which contains for each possible GPU instruction the number of executions per shader instance. Since texture instructions additionally depend on the number of used varying variables (although the impact is smaller compared to vertex shaders), we additionally used $\#V_1$, $\#V_2$, $\#V_3$, $\#V_4$, *numVars*, and *numVarFloats*. Note that one conclusion of our analysis is that executing the very same code in a fragment shader is typically faster than in a vertex shader.

We created the submodel $m_{\text{cmdsFS}}(ctx)$ using "degree=6", "thresh=0.0000002", and specified all components of $\#FS_{cmds}$ in "linpreds" (cf., Sec. 3.1.3.2). As for $m_{\text{cmdsVS}}(ctx)$ (cf., Sec. 3.5.3.4), providing "linpreds" reduces the search space for

MARS and is valid since all evaluations confirm a linear dependency on the number of GPU instructions. A total of 44 terms were selected. The majority of 32 terms with degree 1 express the execution time per GPU instruction. Although the GPU instruction set contains 97 different instructions (cf., Appendix A.1), we were only able to produce about a third of them. This is mainly due to the fact that many instructions such as bit operations or atomic operations are not supported by OpenGL ES 2.0 shaders. The few terms with degree greater than 1 represent the execution time of texture commands depending on the varying variables. As depicted in Fig. 3.16a, the prediction error was below 250 ps for



(a) Cumulative distribution (b) Residuals vs. fitted

Figure 3.16.: Error of submodel for fragment shader commands execution time

95 % of the samples and always below 1.9 ns, which again is a very good result. The residuals vs. fitted plot (Fig. 3.16b) shows that the fitted values are in the range of up to about 170 ns. In few cases, the fitted values of texture commands show linear errors of small magnitude. These errors could be reduced by using a lower threshold. We did not do this since the accuracy of $m_{\text{cmdsFS}}(ctx)$ is sufficient and lower thresholds tend to overfitting.

3.5.3.6. Predicted Texture Lookup Time

In this section, we describe how the submodel $m_{\text{texld}}(shader)$ was obtained, which predicts the overhead of texture lookups exceeding the time needed to lookup a texture of width=1 and height=1. This is due to the fact that the execution time for a texture lookup of size 1x1 is already included in $m_{\text{cmdsVS}}(ctx)$ and $m_{\text{cmdsFS}}(ctx)$. Thus, the submodel $m_{\text{texld}}(shader)$ only represents the overhead caused by texture memory read operations.

Many OpenGL programs use textures to create realistic 3D scenes. Typically, coordinates calculated by the vertex shader are used in the fragment shader to

lookup the corresponding color value stored in the texture. Since the GPU-internal cache for texture data is limited, looking up texture data might imply reading data from memory (system memory or dedicated GPU memory). Especially with embedded GPUs, the caches are small and many lookups imply memory reads. As a matter of fact, the execution time of a texture lookup is not always constant but depends on the ratio of cache hits to read operations. In our evaluations, we observed that memory lookups by a texture unit can run in parallel to the shader code placed after the texture lookup call. However, if the shader code later reads the result of the texture lookup, it seems to be blocked until the result is ready.

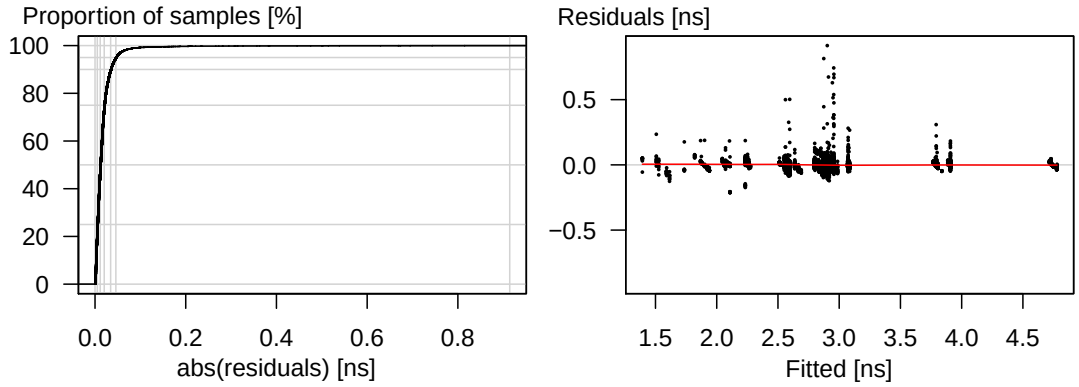
More precisely, the overhead introduced by shader lookups depends at least on the following parameters:

- The exact place of the lookup calls in the shader code
- The size of the texture
- The probability of looking up a cached value, which depends on:
  - the distribution of the lookup coordinates
  - the lookup frequency
  - the exact permutation, if multiple textures are used
  - the position in the shader code the value is first read (if exists)

Creating a model that considers all relevant parameters is unfortunately not feasible. Using other machine learning approaches, like deep learning, would increase both, the required number of samples, and the CPU overhead needed for prediction, by orders of magnitudes. Additionally, the GPU cache hit ratio is not available unless the OpenGL rendering pipeline would be emulated on the CPU, first. To this end, we created a worst-case submodel that assumes that texture lookups occur always at the end of a shader and thus no code runs in parallel (which would decrease the overhead). We created training data that uses between 1 and 3 texture lookups at the end of the shader code. For the texture width and height, 18 different values were used, respectively, creating 324 different texture sizes. The execution time of size 1x1 was used as offset, according to the purpose of m_{texld}. The submodel m_{texld}*(shader)* is used for both, vertex and fragment shaders. To this end, we count the number of texture lookups of the respective shader that is provided to the submodel as *shader*. Since vertex and fragment shaders show a quite similar behavior in this regard

and the overestimation of texture lookups is typically high, a separate submodel is not justified.

We created a MARS submodel for a texture lookup burst of $numTexld$ lookup calls and a texture with width w pixels and height h pixels. We used the features $numTexld$, $log(w)$, $log(h)$, $log(w \times h)$, and $1/(w \times h)$. In many evaluations, this feature set turned out to fit well while keeping the size of the submodel small. A total of 15 terms were selected with $log(w \times h)$ being the most relevant feature. As depicted in Fig. 3.17b, the fitted values were between about 1.4 ns and about 4.9 ns. The residuals show outliers, which are typically positive and caused by



(a) Cumulative distribution

(b) Residuals vs. fitted

Figure 3.17.: Error of submodel for texture lookup calls

interfering memory access. The cumulative distribution, depicted in Fig. 3.17a, shows that 95 % of the samples were fitted with less than 50 ps error. Given that memory access is subject to manifold interference, the achieved accuracy is very good. Using more terms (e.g., created by using a lower threshold) did not provide a significant improvement in accuracy.

3.6. Online adaption

For some rendered scenes, the predicted execution time might lean to either overestimation or underestimation. The concept of online adaption presented in this section is an optional improvement, which detects and corrects that. Online adaption assumes that a prediction deviation that occurs for rendering a certain scene with an OpenGL program, will typically occur also for the rendering of a similar scene in the future, e.g., when rendering the next frame.

The measured execution time of a CG is available after its execution is finished. We described in the Sections 3.5.2 and 3.5.3 how the measured execution time is used to profile performance parameters and to create training data for machine learning models. A further use of the measured execution time is online adaption. This means, we trace for each CG the predicted execution time, and, after the measured execution time is available, the respective correction factor is updated. Each prediction uses the current value of the respective correction factor to improve prediction accuracy.

In Listing 3.4 the code is depicted that runs when the execution time of a CG is predicted, after the other prediction models presented earlier. As shown earlier in Listing. 3.1, the function "online_adaption()" returns the execution time for a CG. Since the concept of online adaption is optional, the "online_adaption()" function returns the prediction execution without adaption in the case that online adaption is not used. If this is the case, the sum of the four different kinds of execution times is returned (Lines 1–2). According to Listing. 3.1, the variables *et_flush*, *et_clear*, *et_sb*, and *et_draw* contain the execution times of our models for Flush, Clear, SwapBuffers, and Draw, respectively.

If online adaption is enabled, we first check all CGs that have finished since the previous execution time prediction (Line 3). CGs with an execution time below *UPDATE_THRESHOLD* are skipped (Line 4), since measurement errors would have a too high impact and changing the correction factor is not justified in such a case. The correction factor that would have predicted the correct value is computed as *tCF* (Line 5). When a 3D application changes the OpenGL program, this often indicates that a different scene or partial scene is rendered. To this end, dedicated correction factors for each program are used. Since different prediction models are used for the different CG types—such as SwapBuffers and Draw—, we additionally use dedicated correction factors for each CG type. We distinguish between the predicted execution time of a CG of its type (*predET_of_type*) and auxiliary execution time (*predET_aux*). We

Listing 3.4: **online_adaption()** (Calculates the predicted execution time of a **CG**, with or without online adaption)

```
 1 if online adaption disabled:
 2    return (et_flush + et_clear + et_draw + et_sb)
 3 for each CG finished in the meanwhile:
 4    if CG.measuredET > UPDATE_THRESHOLD:
 5       tCF = (CG.measuredET  -  CG.predET_aux) / CG.predET_of_type
 6       newCF = CG.program.CF[CG.type] * SMOOTHINGFACTOR
 7                              + (tCF * (1 - SMOOTHINGFACTOR))
 8       if (newCF > CG.program.CF[CG.type])
 9          CG.program.CF[CG.type] =
10                   MIN(newCF, CG.program.CF[CG.type]*CF_MAX_INCREASE)
11       else
12          CG.program.CF[CG.type] = MAX(0.000001,
13                MAX(newCF, CG.program.CF[CG.type]*CF_MAX_DECREASE))
14 nextCG.type = determine_type() // UNKNOWN, EVENT, DRAW, SB
15 nextCG.predET_aux = et_flush
16 nextCG.predET_of_type = 0
17 KF = 1
18 nextCG.program = current_gl_program
19 if nextCG.program:
20    KF = nextCG.program.CF[CG.type]
21    if nextCG.type = DRAW:
22       nextCG.predET_of_type = et_draw
23       nextCG.predET_aux += et_clear + et_sb
24    if nextCG.type = SB:
25       nextCG.predET_of_type = et_sb
26       nextCG.predET_aux += et_clear + et_draw
27 return (nextCG.predET_of_type * KF) + nextCG.predET_aux
```

then use exponential smoothing to calculate *newCF* (Lines 6–7). Exponential smoothing was first presented by Brown [Bro56]. In order to prevent extreme changes caused by single prediction deviations, we extended Brown's concept by restricting changes to the correction factor (Lines 8–13). More precisely, the parameters *CF_MAX_INCREASE* and *CF_MAX_DECREASE* define the maximum allowed change of a correction factor per executed CG.

In the second part, starting in Line 14, we assign the adapted execution time, i.e., using the respective correction factor. To this end, we determine the type of the CG and the current OpenGL program. For the rare case where no OpenGL program is currently used (e.g., at OpenGL initialization), no correction factor applies (Lines 17 and 19). The correction factors are applied only to the respective part (i.e., *predET_of_type*) of the execution time, the remaining time (e.g., pc_{flush}) is assigned to *predET_aux* (Lines 21–27).

Online adaption allows a fine-grained adjustment of the prediction to the rendered scenario and application. The prediction models presented in the Sections 3.4 and 3.5 predict the execution time even for completely unknown applications. These concepts can be combined with our proposed online adaption concept to improve the average accuracy.

3.7. Implementation

We have implemented our concepts for the Freescale i.MX6 embedded platform using a Yocto 1.8 Linux distribution with Linux kernel 3.14. The OpenGL ES Context Monitor and the Predictor with the Prediction models (cf., Fig. 3.6) are implemented in a shared library called "libETP". It consists of 16502 lines of C code and 1847 lines of comments in 42 .c and .h files (determined by "cloc"). The Execution Time Monitor is implemented in kernel space and uses the timestamps taken in the 3D GPU's interrupt service routine (ISR). Since we perform time measurement within an ISR, kernel latency directly affects precision of timestamps. To this end, we applied the preempt-rt patch to the Linux kernel to improve latency. Additionally, the priority of the 3D GPU's ISRISR was increased from 50 to 95, to prevent other threads (including other ISRs) from delaying or interrupting the 3D GPU's ISR.

3.7.1. Architecture

The implementation of our framework extends the existing Linux architecture. The shared library libETP intercepts OpenGL ES 2.0 and EGL calls and performs the prediction. The kernel space driver is extended to measures the execution time of each CG. A bi-directional interface between libETP and the kernel space driver provides the kernel space with the estimated execution time of each CG and informs libETP about the measured (i.e., real) execution time after the GPU finished execution. The architecture of our framework is depicted in Fig. 3.18. 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 for each application thread accessing the GPU. It can be mapped into user space using a *mmap* call on the modules' associated 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 execution time (assigned in kernel space). The segment index represents the array index containing the data for the arriving CG. The user space part of the prediction is encapsulated in the shared library libETP, which is binary compatible to libGLESv2 (for OpenGL ES 2.0) and libEGL (for EGL) libraries. It intercepts all API calls of these libraries and traces them within the prediction module (❶ in Fig. 3.18) to keep track of the current OpenGL ES 2.0 Context. Subsequently, it forwards the calls to the native GPU driver libraries.

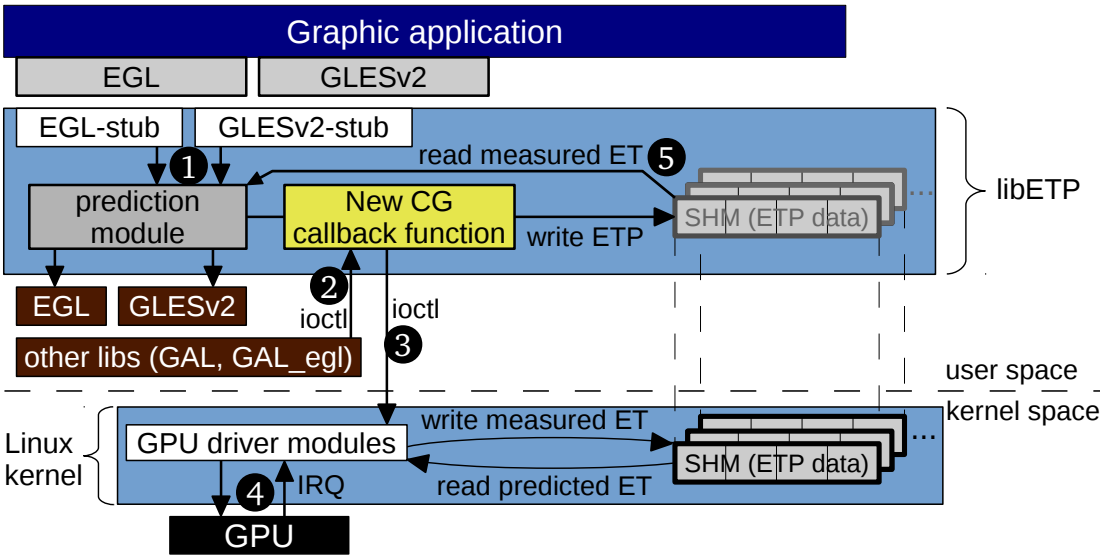


Figure 3.18.: Implemented framework architecture

Eventually, the native GPU driver wants to flush the current CG to the kernel, which is implemented by calling the "ioctl" function. In libETP we intercept the call to "ioctl" and invoke a callback function (❷). The callback function completes the current DRAW batch, calculates the predicted execution time, and writes it to the next unused shared memory segment. Finally, the callback function invokes the native "ioctl" function with the respective parameters (❸). Each time the GPU driver module in kernel space receives a syscall with a new CG, it reads the predicted execution time from the associated shared memory ETP data. Thus, a GPU scheduler can use the predicted execution time for scheduling decisions (see Chapter 4). Before sending a CG to the GPU (which is labeled as ❹), the kernel GPU driver always appends a GPU operation—similar to what was proposed in [KLRI11]—which causes the GPU to send an interrupt after that CG has been executed. The ISR in the GPU driver module takes the current time stamp, calculates the CGs execution time, and writes it to the ETP data. This allows the prediction module to read the execution times from the ETP data (❺). Next, we describe in more detail how libETP is implemented, and especially how the prediction module works.

3.7.2. Initialization of the shared library libETP

When the application issues its first EGL or OpenGL ES 2.0 command, libETP performs the following steps:

- Read prediction parameters, such as prediction model, or evaluation modes

- 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 shared memory (ETP data) from the kernel module

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

3.7.3. Prediction model creation

In order to create the offline models for the system-specific performance parameters and the MARS models, we created a small interface for evaluation applications. More precisely, libETP exports the following functions, which can be accessed by applications running on top. The function "getCurrentDataIndex" returns the next index of the SHM ETP data. It is called before and after one or more CGs have been submitted (e.g., by calling glFlush). These two calls provide the range of ETP data indices and are passed to the function "getMeasuredExecutionTimeOfRange", which sleeps until all CGs have finished execution and returns the aggregated values of the measured execution time and the measured number of fragments. Since the MARS models must not contain the system-specific performance parameters, these parameter values can be queried from libETP using the "getConstPredictionParameters" function.

To be aware of the current render buffer resolution, libETP uses the first call for eglInitialize to keep the current EGLDisplay handle. 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.

For profiling during runtime, a dedicated window surface and OpenGL ES 2.0 Context are created. For the profiling we use a render buffer size of 1024 by 768 pixels. When an OpenGL program is used the first time for rendering, depending on the prediction mode, MARS-based or profiling-based models are used. If profiling during runtime is used, libETP checks whether an XML settings file exists that contains the matching profiling-based performance parameters. An example of such a file is depicted in Appendix A.2. If the XML settings file exists and contains the profiling parameters for this particular OpenGL program, those

cached values are used and the profiling during runtime must not be executed again. Otherwise, the profiling during runtime as described in Sec. 3.5.2.2 is executed. For the profiling of the execution time of vertex operations t_{VS} we draw 30000 vertices that are rasterized to 0 fragments. For profiling the execution time of fragment operations t_{FS} we draw 3 triangles, each covering half of the render buffer, we therefore render 3×393216 fragments.

When the Vivante GPU driver receives an eglSwapBuffers call, it does not directly insert the respective GPU instructions into the command buffer. Instead, the GPU driver first flushes the staged GPU instructions as a CG and subsequently inserts the GPU instructions that perform the SwapBuffers operation. This means that the first CG submitted after an eglSwapBuffers call actually does not contain any SwapBuffers instructions, but the subsequent CG. This is due to the fact, that GPU drivers realize SwapBuffers typically in dedicated CGs.

In our implementation for Vivante, the execution time of a buffer resolve command is not estimated by the render buffer size as provided by the OpenGL ES Context Monitor, but by directly tracing the corresponding buffer resolve commands in the command buffer. Whether a CG is of type "Draw" or of type "SB" (which is relevant for the online adaption concept presented in Sec. 3.6), can easily be determined by taking the GPU driver behavior upon a "eglSwapBuffers" call into account. When a "eglSwapBuffers" call is detected by the OpenGL ES Context Monitor, the next CG is of type "Draw", followed by a CG of type "SB".

3.7.4. Used libraries and algorithms

Our libETP shared library depends on the following shared libraries:

EGL: The native GPU driver for EGL.

GLES2: The native GPU driver for OpenGL ES 2.0.

xml2: An open source library providing XML support.

m: The standard math library.

rt: The real-time extensions library.

As described in Sec. 3.5.1.1, we calculate the size of the projected triangle. Since a triangle potentially intersects with the order of the render buffer, we calculate

the intersection, i.e., the area of the triangle clipped to the boundaries of the render buffer. To this end, we use the Sutherland-Hodgman clipping algorithm [SH74].

3.7.5. Modes of operation

A couple of parameters are supported by libETP that can change the default behavior. Next, we briefly describe the parameters used in our evaluations:

MODE (default "BB") selects the prediction model:

- **BB** bounding box heuristic with profiling-based shader prediction
- **TS** triangle samples heuristic with profiling-based shader prediction
- **MARS** bounding box heuristic with MARS-based shader prediction
- **COMPARE** Run the three previous modes in parallel and dump results. Additionally, the history-based prediction is executed for comparison.

PROFILING_FB_NUM (default 3) Number of the framebuffer used for profiling.

OVERPREDICT_FACTOR (default 1.0) The predicted execution time is multiplied with this factor before passing it to the kernel space.

REAL_NUM_FRAG_MODE (default "no") Evaluation mode that allows for a fair comparison between profiling and MARS. If enabled, libETP ensures that each CG contains not more than one DRAW batch. The prediction of this DRAW batch is performed *after* the CG has finished execution and is aware of the real number of fragments, measured by the GPU.

EXIT_AFTER_NUM_DRAWS (default ∞) The process is terminated after the specified number of DRAW CGs was rendered.

PROFILING_XML_FILE (default "~/libetp_profiling.xml") Full path to the XML profiling data file. Each application process uses its own XML file.

3.8. Evaluation

In this section, we present the evaluation setup and discuss the results.

3.8.1. Setup

3.8.1.1. System

For the evaluation of our execution time prediction concepts we used the implementation described in Sec. 3.7. We also implemented the history-based execution time prediction approach proposed by Kato et al. [KLRI11], which serves as a comparison to our approaches. As hardware platform we used a Freescale i.MX6 SABRE Automotive Platform. The board features a quad core ARM CPU running at 800 MHz, 2 GB of RAM, and a Vivante GC2000 GPU for 3D rendering. We used a Yocto 1.8 system image based on Linux kernel 3.14.28 patched by Freescale with preempt-rt patches and the Vivante driver V5.0.11.p4.25762. For evaluation we used the FIFO algorithm of our GPU scheduler (cf., Chapter 4), which does not synchronize to the screen's *refresh rate* and therefore does not slow down the applications. For more details on the platform and the GPU scheduler, see Sec. 4.5.

3.8.1.2. Measurement accuracy

Accurate time measurements of the GPU execution are very important. Measurement errors during profiling or at creating training data can lead to wrong predictions. Since the timestamps of finished CGs are taken in the 3D GPU's ISR, a delayed execution of the ISR would directly affect measurement accuracy. Measurement errors also add inaccuracy to measurements of the prediction error. Execution of an ISR by the operating system can be delayed due to context switches or non-interruptible kernel code. To measure this latency of the Linux kernel, commonly the tool "cyclictest" of the rt-tests tool suite [rt-18] is used. In Figure 3.19, we depict the Linux kernel latency distribution—summarized over all four CPU cores—created by "cyclictest"[6]. The red curve shows the Linux kernel latency, if only cyclictest was running. An average latency of about 20 µs and a maximum of 50 µs was observed. These are actually quite good results, which are even significantly better than the results observed by OSADL on a Vanilla kernel with preempt-rt patches [OSA], where

[6] Used parameters: "cyclictest -l100000000 -m -Sp99 -i200 -H2000 -q"

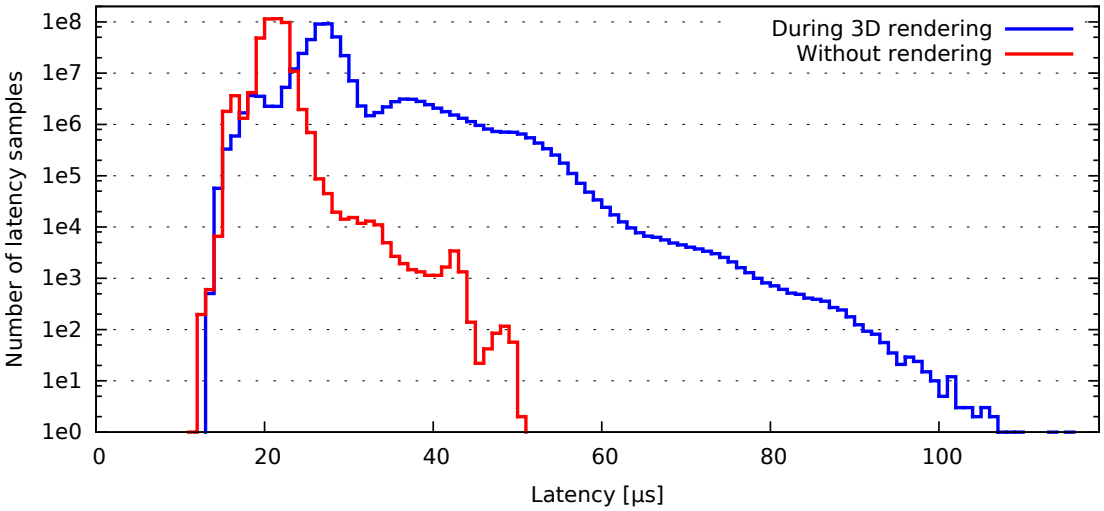


Figure 3.19.: Kernel latency distribution

the average and worst-case latencies are significantly higher. Thus, the latency of the Linux kernel version published by Freescale is much better, yet. We additionally evaluated the latency while the 3D GPU driver was under heavy load and depicted the results as blue line in Fig. 3.19. We executed 20 instances of the small "es2gears" benchmark program and activated FIFO scheduling, which just dispatches the small CGs as fast as possible. The 3D dispatching was active during the whole time cyclictest was active (about 5.5 hours). According to our results, dispatching 3D CGs has a significant impact on the latency, which increases to about 26 µs on average and 115 µs in the worst-case. This shows that the used implementation of dispatching and ISRs leave much room for improvement concerning low latency. Actually, the latency measured by cyclictest is the latency for user space processes, which, e.g., includes the time for the CPU scheduler to select the process and the time needed for the context switch from kernel to user space, which explains the minimum latency of about 11 µs. Since the time critical components of our GPU scheduling framework are all implemented in kernel space, especially the overhead for context switch to user space does not apply, and the minimum latency might be a bit smaller than what was observed with cyclictest. Nevertheless, our results indicate that the measurement accuracy on the used platform is limited to the order of tens of microseconds. This directly affects our measurements of the execution time of CGs in Sec. 3.8.5, including the effectiveness of online adaption. Thus, the limited accuracy of the Linux kernel shows up as a prediction error in our measurement results that cannot be avoided.

3.8.1.3. System-specific parameters

We calculated the system-specific performance parameters (cf. Sec. 3.3) using linear regression. The training data was created by rendering 10000 triangles with 2 vertex attributes of size vec4[7]. The number of DRAW calls was varied between 1 and 10000. Since the driver's command buffer is limited in size, higher numbers of DRAW calls result in multiple CGs. For instance, in the evaluation data, 10000 DRAW calls were split up by the driver into 10 CGs. Using linear regression, we obtained the two parameters

$$pc_{\text{flush}} = 34.49\,\mu\text{s}$$
$$pc_{\text{drawconst}} = 0.421\,\mu\text{s}.$$

The system constant for the SWAPBUFFERS command was determined using the rounded average of the evaluations described later in this section as

$$pc_{\text{swapbuffers}} = 1.85\,\text{ns}.$$

For the used hardware architecture, our evaluations determined

$$pc_{\text{clear[btypes]}} = 0\,\text{s}$$
$$pc_{\text{depth}} = -0.474\,\text{ns}$$
$$pc_{\text{blending}} = 1.382\,\text{ns}.$$

3.8.1.4. Online adaption parameters

Our online adaption algorithm (cf., Listing 3.4) uses three parameters that affect smoothing. We compared the prediction error distributions of different combinations of these parameters. Based on the observed error distributions, we used the following default parameters in our evaluations.

$$SMOOTHINGFACTOR = 0.9$$
$$CF\_MAX\_INCREASE = 1.1$$
$$CF\_MAX\_DECREASE = 0.9$$

[7] Increasing the number of vertex attributes increased the amount and magnitude of outliers but did not reveal as relevant feature.

3.8.1.5. Applications

For our evaluations, we used a wide choice of OpenGL ES 2.0 applications. In Sec. 3.8.4, we used a full run of the benchmark glmark2-es2, provided by [glm]. In Sec. 3.8.3 and Sec. 3.8.5, we used the applications listed and briefly described in the following. In Fig. 3.20, a screenshot for each application is depicted.

(a) es2gears, a demo provided by MESA [mes]. Uses 280 DRAW commands and 1360 vertices to render one frame that consists of 3 rotating gearwheels.

(b) glmark2-es2 "build": The "build" scene rotates a given 3D model, by default a horse containing 21516 vertices.

(c) glmark2-es2 "shading": The "shading" scene uses a more sophisticated fragment shader for improved lightning and rotates a cat model with 43044 vertices.

(d) glmark2-es2 "texture": Maps a texture on the 3D object, by default a simple cube consisting of 36 vertices.

(e) Automotive speedometer application (resolution 456x456): Uses two textures, each mapped on a square of two triangles, to display an indicator needle showing the current speed. It uses about 260000 fragments.

(f) Quake 3 "demo four" (resolution 1440x544): OpenGL ES 2.0 port of the open source game ioquake3[8] (resolution 1440x540).

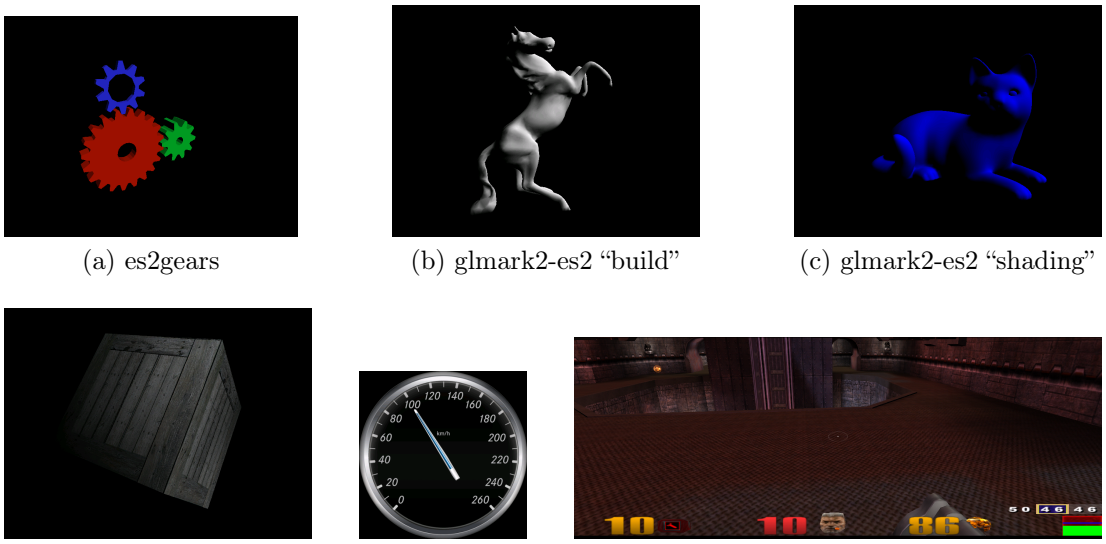
Additionally, a comparison of the selected applications is provided in Table 3.5. Our selection of applications consists of existing applications that are

Table 3.5.: Comparison table of 3D applications used for evaluation

	Number of			Shader	Uses				window
Application	draws	vertices	scenes	complex.	textures	blending	VBOs	MVPM	size
es2gears	high	medium	3	low	no	no	no	yes	medium
glmark2-es2 "build"	low	medium	1	medium	no	no	yes	yes	medium
glmark2-es2 "shading"	low	high	1	medium	no	no	yes	yes	medium
glmark2-es2 "texture"	low	low	1	high	yes	no	yes	yes	medium
speedometer	low	low	2	medium	yes	yes	no	no	small
Quake 3 "demo four"	medium	high	many	medium	yes	yes	no	yes	big

publicly available. The only exception is the speedometer application, which was written for the ARAMiS demonstrator (cf., Sec. 2.3), and intentionally also

[8] `git://github.com/libv/ioquake3;branch=limare;rev=033a8a09546fcbc07e71accd6381191089e8bfe8`



(a) es2gears (b) glmark2-es2 “build” (c) glmark2-es2 “shading”

(d) glmark2-es2 “texture” (e) speedometer (f) Quake 3 “demo four”

Figure 3.20.: Screenshots of evaluated applications

covers the uncommon but valid case that no MVPM is used. Since it is not possible to cover all possible combinations of 3D application features, we selected these 6 applications that represent combination and let us perform a quite diverse and representative evaluation of our execution time prediction concepts.

If not stated otherwise, the resolution is 800x600. All applications are patched to use the platforms native framebuffer devices instead of X11. We evaluated for application 10000 CGs using the COMPARE mode of libETP. The COMPARE mode executes the implemented prediction methods in parallel and dumps it in a separate file. Since writing to this file is flushed after each completed CG and due to the amount of written data–in particular a hex dump of the command buffer content–the applications are significantly slowed down. This, however, only applies to the COMPARE mode, while in normal operation the prediction introduces only little overhead.

3.8.2. Coverage factor

As explained in Sec. 3.5.1.4, the fragment heuristic based on bounding boxes uses 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. Since the coverage factor significantly influences the accuracy of the respective fragment heuristics, we evaluated the coverage for the

applications presented in Sec. 3.8.1.5. Table 3.6 shows for these applications the average area covered by the bounding boxes. Additionally, the measured number

Table 3.6.: Comparison of the measured number of fragments with the area covered by bounding boxes

	Average number of fragments		
Application / scene	Measured	of the BBs	Ratio
es2gears	44585	70914	62.9 %
glmark2-es2 “build”	51349	184078	27.9 %
glmark2-es2 “shading”	91011	225935	40.3 %
glmark2-es2 “texture”	157292	157292	100.0 %
speedometer	129959	146878	88.5 %
Quake 3 “demo four”	62929	157189	40.0 %

of fragments is depicted, which allows to calculate the ratio between the size covered by the bounding box and the actual number of fragments, i.e., the best coverage factor for each application. This mix of application has an average ratio of about 60 %. Therefore, for our evaluations we used a coverage factor of 0.60. We are aware, that this might not be accurate for all scenarios or applications. However, the coverage factor targets a good estimate for applications unknown so far. After an application has rendered a couple of CGs, its prediction accuracy can be improved using the actually measured execution time and online adaption (cf. Sec. 3.6).

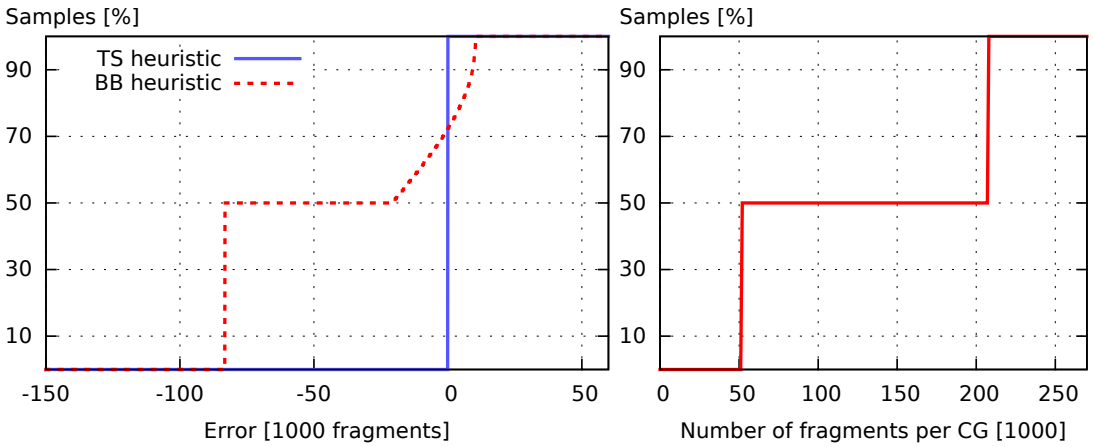
3.8.3. Fragment Heuristics

The number of fragments has a very high impact on the execution time. In this section, the accuracy of our bounding box (BB) heuristic is evaluated and compared to our triangle samples (TS) heuristic. We compare both values with the real number of fragments as provided by the Vivante profiler[9].

3.8.3.1. Speedometer application

The results of the fragment heuristics for the speedometer application are depicted in Fig. 3.21. Each frame is rendered using two CGs. The first CG draws the background texture (207 936 fragments), the second CG draws the indicator texture (about 52 000 fragments), as observed in Fig. 3.21b. For the

[9]We use the Vivante profiler field “ra_valid_pixel_count”



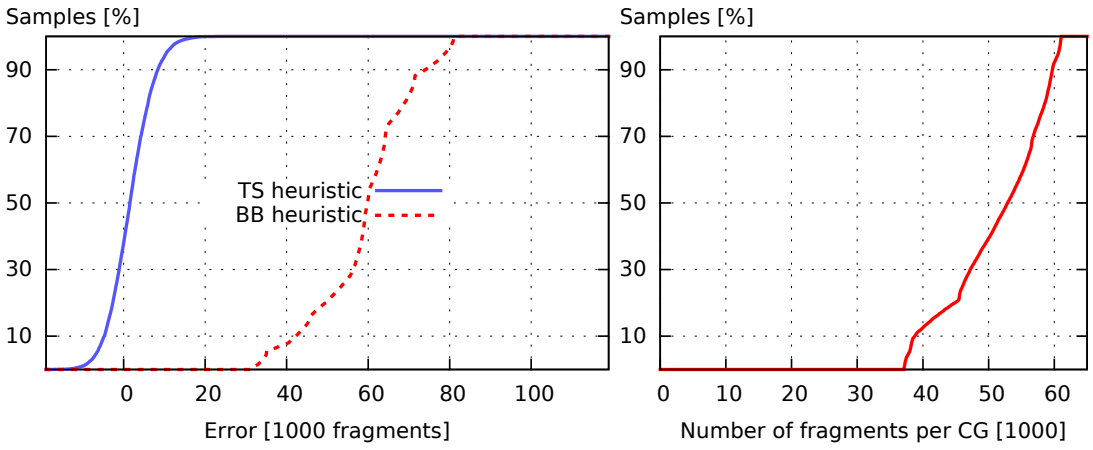
(a) Error distribution of prediction (b) Measured number of fragments

Figure 3.21.: Accuracy of fragment heuristics, speedometer application

background CG, the BB approach determines the exact area of the texture, but since the coverage factor 0.6 is applied, it is off by 40 %, cf. Fig. 3.21a. Thus, the BB approach underestimates every second CG by about 104 000 fragments. The underestimation of the foreground indicator texture depends on its angle since the speedometer application performs the multiplication with the model view projection matrix on the CPU instead of the GPU. Therefore, the BB approach creates bounding boxes of different sizes depending on the angle, which results in an error between −20 631 fragments and 10 392 fragments (i.e., −39.7 % to 20.0 % of the size of the indicator texture). The TS approach, which does not use a coverage factor, predicts always the almost exact number of fragments for both CGs. The highest error observed with the TS approach was 66 fragments (0.13 %) and is caused only by the small error introduced by our triangle size approximation (cf., Sec. 3.5.1.1).

3.8.3.2. Glmark2 benchmark application

The results of the fragment heuristics for the glmark2 “build” benchmark are depicted in Fig. 3.22. Each frame is rendered using only one Draw command that is contained in one CG. As depicted in Fig. 3.22b, the rotation varies the number of fragments between 37 164 and 61 098. For this specific application, the average ratio between bounding box area and number of fragments is 0.279 % (cf., Table 3.6), since much space of the bounding box created by the horse model is not covered by any fragment. Therefore, the used coverage factor of 0.6 is too high. As observed in Fig. 3.22a, this leads to an average error of the BB approach



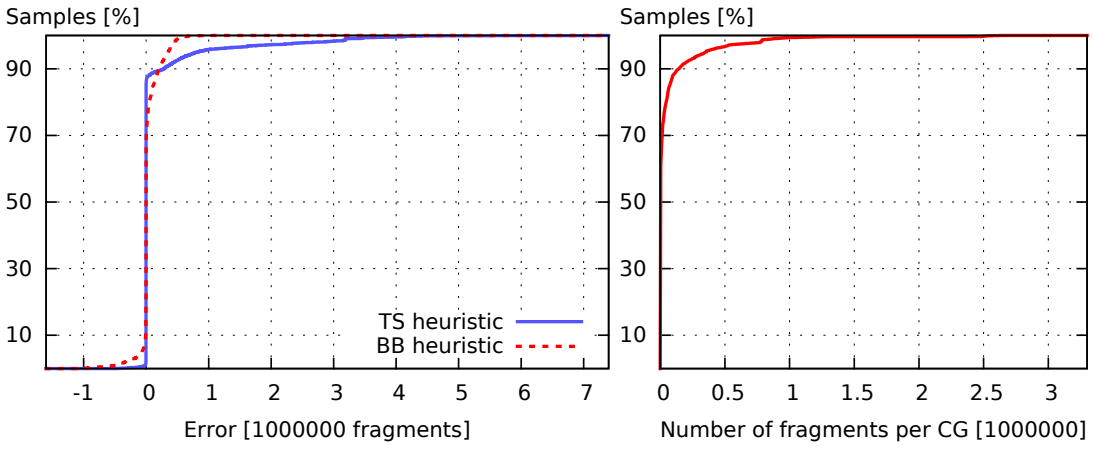
(a) Error distribution of prediction (b) Measured number of fragments

Figure 3.22.: Accuracy of fragment heuristics, glmark2-es2 "build" benchmark

of 220 % (59 772 pixels). The error of the TS approach is quite small and due to random selection of the samples from the 7172 triangles of the horse model with the probability $p = 2^{-5} = 3.125\,\%$. The TS approach shows very good average accuracy, except for a minor overprediction caused by small, overpainted regions in the horse model.

3.8.3.3. Quake 3 "demo four" application

The results of the fragment heuristics for the Quake 3 "demo four" are depicted in Fig. 3.23. Fig. 3.23b shows that many CGs produce very few fragments, while few CGs produce many fragments, up to a maximum of about 3.25 million. As



(a) Error distribution of prediction (b) Measured number of fragments

Figure 3.23.: Accuracy of fragment heuristics, Quake 3 "demo four" application

depicted in Fig. 3.23a, the worst-case overestimation of the TS approach is about 7 327 502 fragments, for BB it is up to about 7 300 000 fragments. Additionally, BB underestimates up to 1.6 million, while TS only up to 0.9 million. For the majority of CGs, both heuristics are very good. However, these CGs are easy to predict since they have only few fragments. A significant challenge with Quake 3 is the effect of the early depth test, which does not create a fragment that will not pass the Depth test, anyway. On the other hand, unfortunate order of the vertices can result in a high overpainting rate, which explains why up to 3.25 million fragments can be rendered to a render target that has a size of only 777 600 pixels. This is the main reason, why the BB approach underestimates about 10 % of the CGs. On the other hand, the TS approach rarely underestimates but significantly overestimates about 10 % of the CGs, since TS, simply speaking, does not consider that early depth test is used. A challenging application such as Quake 3 shows the limitations of heuristics predicting the number of fragments. It also demonstrates that the TS approach has very little underestimation but often overestimates, which is less critical for real-time scheduling.

3.8.3.4. Summary of Fragments Heuristics

The evaluations show that TS prevents underestimation very well but suffers from overestimation if the GPU uses the early depth test to reduce the number of fragments. The BB approach is a lean alternative, which often provides a good accuracy and has no clear tendency to either overestimation or underestimation. In general, for complex multi-layered 3D scenes, especially of untrusted applications, TS is preferred in favor of overprediction. For single-layer 3D scenes, the BB approach is often the better choice, especially since the vertex shader must be emulated for only 8 vertices, instead of up to 256 (for huge models even more) with TS.

3.8.4. Shader execution time

Predicting the execution time of vertex and fragment shader instances is a major prerequisite for a good overall execution time prediction. We presented a profiling-based concept in Sec. 3.5.2 and a machine-learning-based approach using MARS in Sec. 3.5.3. In this section, we show the effectiveness and accuracy of both prediction concepts. To this end, we executed the OpenGL ES 2.0 benchmark glmark2-es2 [glm] using its default mode, which sequentially runs all benchmark scenes. In order to achieve a fair comparison with the

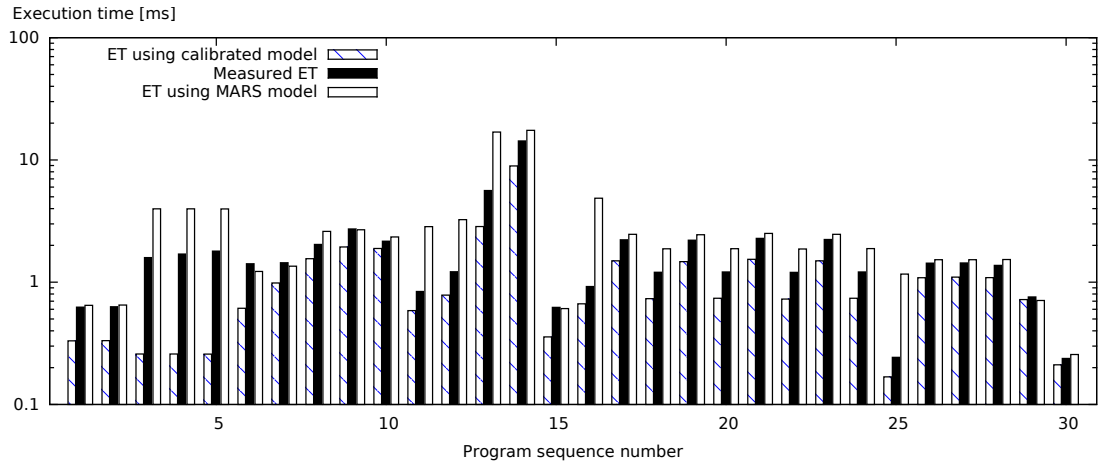


Figure 3.24.: Accuracy of shader execution time prediction concepts

measured execution times, we provided both approaches with the accurate number of fragments, which is provided to libETP by the kernel space driver. To this end, we run libETP in the REAL_NUM_FRAG_MODE mode (cf., 3.7.5), where the CGs are first executed and prediction is done actually after the execution has finished and the real number of fragments is available. In Figure 3.24, the results for the first 30 shader programs are depicted. Further shader programs show similar behavior. The occasional overestimation of MARS is due to the following reasons:

- In the training data for the MARS model for texture load commands (m_{texld}) all texture load commands were at the very end of the shader code. However, if other commands follow, the memory read operations actually execute in parallel, which reduces the overall execution time compared to m_{texld}.

- Texture lookup coordinates of the MARS training data were fully random, which makes the texture cache as inefficient as possible. However, for typical scenarios, close-by fragments are more likely to read the same texture coordinates.

The rare cases where MARS underestimates are caused by:

- Memory operations that could not run concurrently in some parts of the shader code (observed at the sequence numbers 6, 7, and 29) caused an error of up to about 13 %. Without a sophisticated prediction model for concurrent memory access speed, this error cannot be avoided.

The MARS-based approach rarely underestimates and only by small percentage. However, the worst-case assumption used for m_{texld} causes high overestimations in some of the cases where texture lookup commands occur. Nevertheless, the MARS-based submodel is a competitive alternative to the profiling-based approach.

The profiling-based approach typically underestimates. This is mainly due to the fact that the parameters during profiling differ from the real execution. E.g., for profiling a fixed-size texture is used, since the actually used texture size is not known at profiling time and profiling all combinations of texture sizes is not feasible during runtime. To reduce underestimation, using a large texture size during profiling would change this—with the consequence that the profiling-based approach would also tend to overestimation, as it is the case with the MARS-based model. In our evaluations, we nevertheless observed that the MARS-based approach is more effective in preventing underestimation while the profiling-based approach is more effective in preventing overestimation.

The MARS-based approach has the huge advantage that no profiling during runtime is needed. For new applications, profiling would have to be done while other (perhaps critical) applications are executed. To prevent impact, the execution time of profiling CGs must also be known in advance, which would be hard to achieve. Additionally, profiling consumes many GPU resources, for instance some of the GLSL programs took more than 10 s for profiling. If other important applications are running during profiling, the GPU could not be fully utilized by profiling, which would cause the profiling to run much longer—severely impacting usability. The MARS model thus often provides a better, both, efficient, and accurate method to estimate the shader execution time.

3.8.5. Command Group prediction

In this section, the evaluation results for the prediction accuracy of complete CGs under realistic conditions are presented. We compare the profiling-based shader model with the two available fragment heuristics BB and TS. For the sake of clarity, the MARS-based shader model is evaluated with the BB heuristic, only. Thus, TS can be compared to BB, and a profiling-based approach can be compared to a machine-learning-based approach. Additionally, the evaluations show also the effect of activated online adaption (cf. Sec. 3.6). In general, the results without online adaption represent the accuracy of new

applications or scenes, while active online adaption represents situations, where a scene has already been rendered for a couple of frames.

3.8.5.1. Es2gears application

In Figure 3.25, the evaluation results for es2gears are depicted. Most CGs take

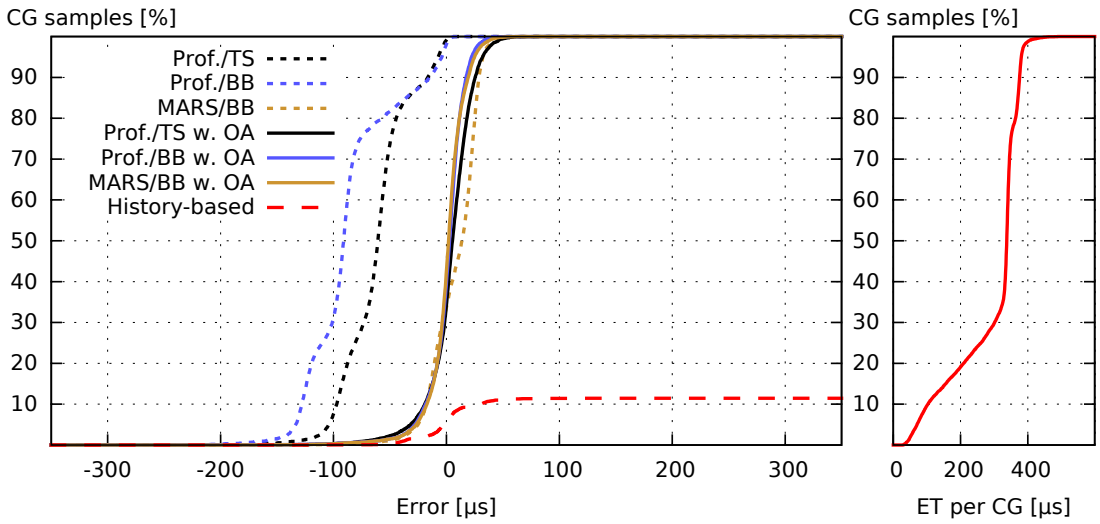


Figure 3.25.: Accuracy of DRAW prediction, es2gears application

about 375 µs. The number of fragments were estimated by the BB approach with an average error of only −4.5 %, while the TS approach had an average error of 32.8 %. This explains why the profiling-based TS approach estimated smaller execution times than the profiling-based BB approach. Since the BB-based fragment heuristic is very accurate for this application, the underestimation of the profiling-based BB approach is due to insufficient accuracy of the profiling-based shader model. In contrast, the MARS-based shader model, which is based on a huge amount of training data, achieves great accuracy, even without online adaption. If online adaption is used, all approaches are very accurate. The history-based approach predicted only about 12 % of the CGs using its history, for the other about 88 % of the CGs, the maximum execution time was predicted. Since the execution time of the CGs containing the SWAPBUFFERS command were much higher (up to 1350 µs), their maximum execution times were used for about 88 % of the CGs.

3.8.5.2. Glmark2-es2 "build" benchmark

In Figure 3.26, the evaluation results for the glmark2 "build" scene are depicted. The rendered horse model is overpredicted by the MARS model by about 30 %,

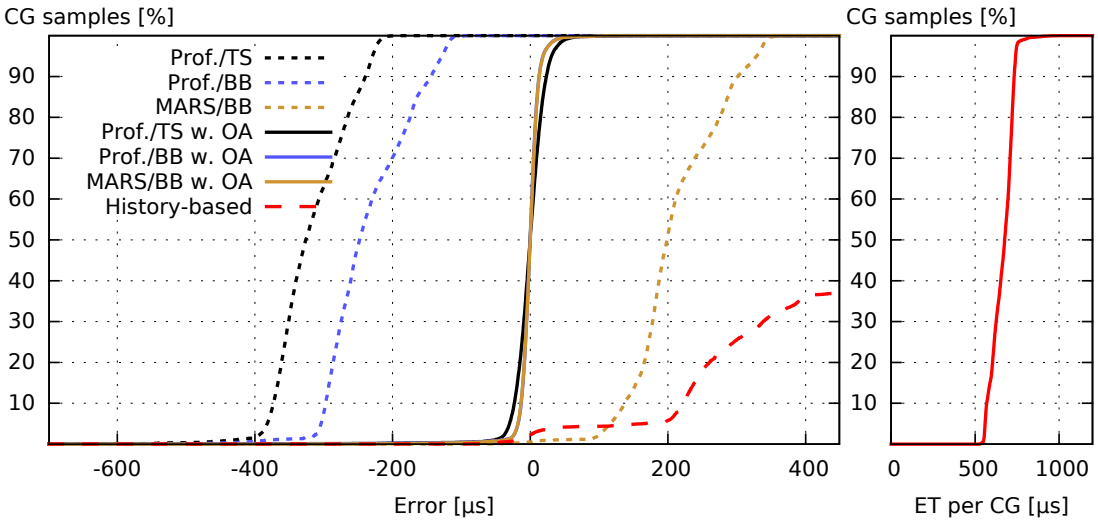


Figure 3.26.: Accuracy of DRAW prediction, glmark2-es2 "build" benchmark

since the bounding box is a too big and therefore the coverage factor 0.6 is significantly too high (a coverage factor of 0.279 would be perfect in this case, see Table 3.6). The profiling-based shader model massively underestimates the execution time per shader instance (cf., program sequence number 1 in Figure 3.24). Although the TS approach overpredicts the number of fragments by only about 3 %, the error of the profiling-approach leads to a significant underestimation of the profiling-based TS approach of about 45 %. The profiling-based BB approach underestimates about 10 % less since the overestimation of the BB heuristic compensates the shader execution time error to some degree. The triangle samples approach has a higher jitter in the prediction of the number of fragments, since on average only about 2.8 % of the triangles are used to extrapolate the number of fragments. Since jitter cannot be compensated by online adaption, the standard deviation is significantly higher than for the bounding box approaches. With online adaption and the profiling-based bounding box approaches, less than 1 % of the samples were an underestimated by more than 42 µs and less than 1 % of the samples were overestimated by more than 38 µs. Table 3.7 shows further error rates, showing that significant prediction errors rarely occurred.

Table 3.7.: Prediction errors of Glmark2-es2 “build”

Prediction mode	Error < 200 µs	Error < 100 µs	Error > 100 µs	Error > 160 µs
Prof./BB w. OA	0.15 %	0.36 %	0.13 %	0.00 %
Prof./TS w. OA	0.17 %	0.39 %	0.16 %	0.00 %
MARS/BB w. OA	0.11 %	0.34 %	0.13 %	0.00 %

3.8.5.3. Glmark2-es2 “shading” benchmark

Figure 3.27 depicts the evaluation results for the glmark2 “shading” scene. Without online-adaption, the profiling-based approaches underestimate significantly by about 50 %. The underestimations of the profiling-based approaches are caused by an imprecise profiling that uses different vertex attribute values than the real application. More precisely, the vertex shader uses the vertex attributes “position” and “normal”, where “normal” are used as normal coordinates to calculate the fragment color. Our profiling does not know the semantic of each vertex attribute and therefore the same constant dummy data is used for all vertex attributes (cf. Sec. 3.5.2). For this shader in particular, the “Color” varying variable typically has the same value. The GPU seems to detect that the fragment shader code is supposed to be executed with the very same input data multiple times and applies a live optimization by executing the fragment shader only once for multiple fragments and then duplicating the result. Unfortunately, this effect makes profiling execute significantly faster than the real application. The bounding box heuristic with profiling is slightly better, since it slightly overestimates the number of fragments. The MARS-based approach also underestimates the shader execution time, but only by a few percent and therefore is much better than profiling. Looking at the results for active online adaption, the two bounding box approaches are able to keep the error very low. In contrast, for the TS-based approach the online adaption still shows a significant error. This is because the triangle samples are selected from the 14348 triangles with p_min, where the set of selected samples differs between different Draw calls. This introduces jitter to the estimated number of fragments that cannot be compensated by online adaption. Since the scene (a rotating cat)—as well as the bounding box—changes slowly, the bounding-box heuristic is smoothed by online adaption better than the TS approach.

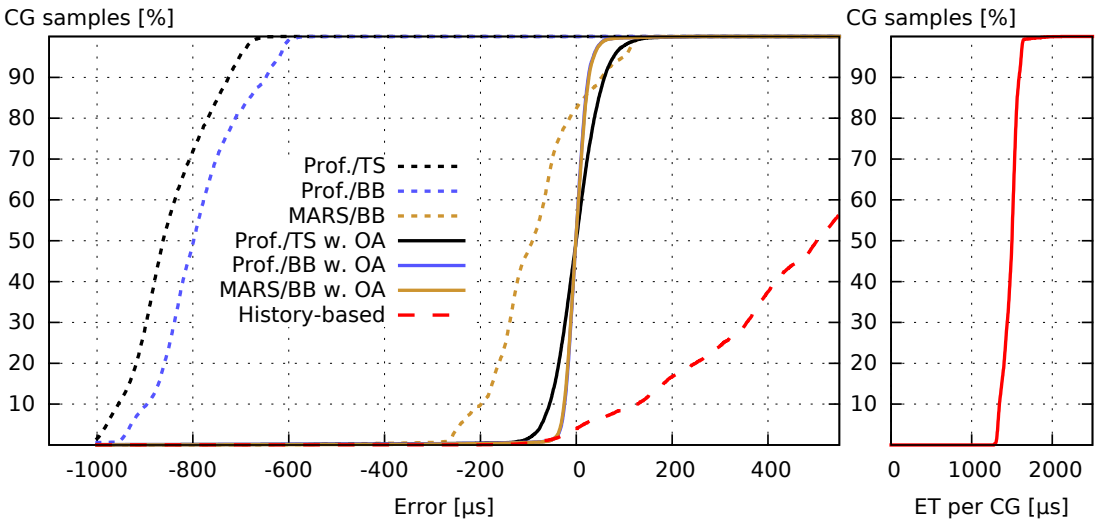


Figure 3.27.: Accuracy of DRAW prediction, glmark2-es2 "shading" benchmark

3.8.5.4. Glmark2-es2 "texture" benchmark

In the glmark2 "texture" scene, the profiling-based approaches without online adaption show a huge underestimation, cf. Fig. 3.28. The "texture" benchmark uses a simple cube model with a high-resolution texture. Since the bounding box approach fits the rendered 3D model but uses a coverage factor of 0.6, the estimated number of fragments is almost exactly 60 % of the real number of fragments, i.e., underestimated by 40 %. The triangle samples heuristic is very accurate: its worst-case error was only 134 fragments (0.085 %). This explains, why the error of the Prof./BB approach is about 90 µs higher than the error of the Prof./TS approach. The error of the profiling-based approaches is caused by the parameters that differ between profiling and real execution. In particular, during profiling textures of a fixed size are used (cf., Sec. 3.5.2) that are—in this case—much smaller than the one used in the "texture" benchmark. This demonstrates a strength of the MARS approach, which uses the texture size of texture lookup commands for prediction. However, since the texture lookup model of MARS is a worst-case estimation (cf., Sec. 3.5.3.6), MARS shows an overestimation of roughly 700 µs. Using online-adaption, all approaches achieve similar, much better prediction accuracy. However, for the "texture" benchmark scene, online adaption is less effective than for many other applications (such as, e.g., the "shading" benchmark scene discussed in the previous section). This is due to the fact that at the 3D cube that is rendered for the "texture" scene, the number of fragments changes quite fast when a cube side surface becomes

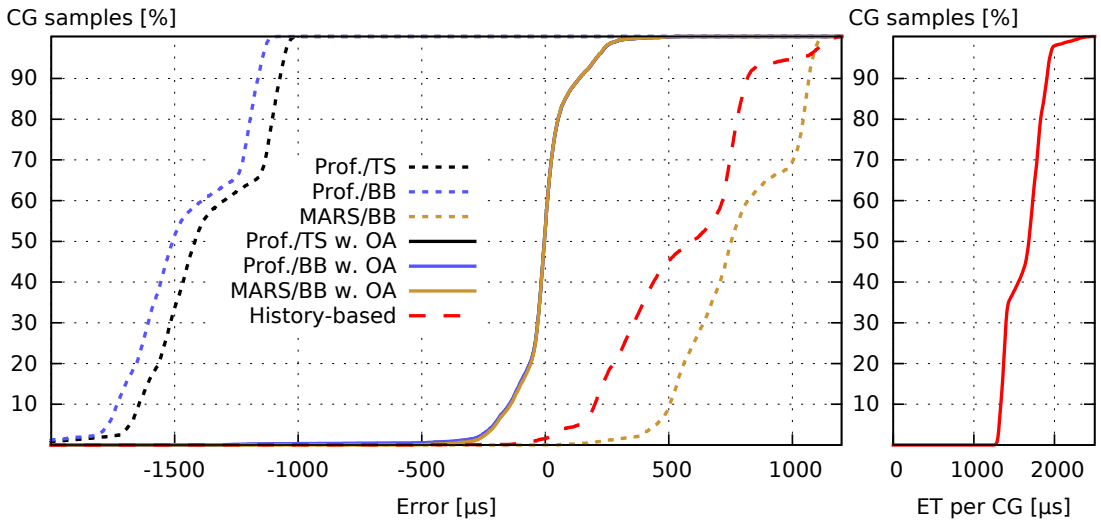


Figure 3.28.: Accuracy of DRAW prediction, glmark2-es2 "texture" benchmark

visible. Depending on the viewing angle, the cube has at least 147 912 fragments and at most 164 680 fragments. For many other applications, such as the cat model rendered for the "shading" scene, the number of fragments changes over time more slowly. The history-based approach was able to achieve a cache hit in only 19 out of the 10 000 CGs and thus almost always predicted the highest execution time observed.

3.8.5.5. Speedometer application

The results for the speedometer application are depicted in Fig. 3.29. The rendering of the speedometer combines two scenes: the background texture and the pointer texture indicating the speed. The two scenes are rendered in separate CGs of similar execution times. However, the CG drawing the pointer texture has about 25 % of fragments compared to CG drawing the background texture, which results in two classes of CGs. While the TS heuristic quite accurately predicts the number of fragments (maximum error below 0.13 %), the BB heuristic underestimates the CG drawing the background texture by 40 % due to the coverage factor of 0.6. This explains the lower error of the profiling-based TS approach for 50 % of the samples. The two textures used for the speedometer have a high resolution, significantly higher than the resolution used while profiling. This causes the profiling-based approaches to underestimate the fragment shader execution times. The MARS-based approach overpredicts the texture operations due to its worst-case model. Since the two classes of CGs differ by their amounts of fragments, MARS overpredicts the

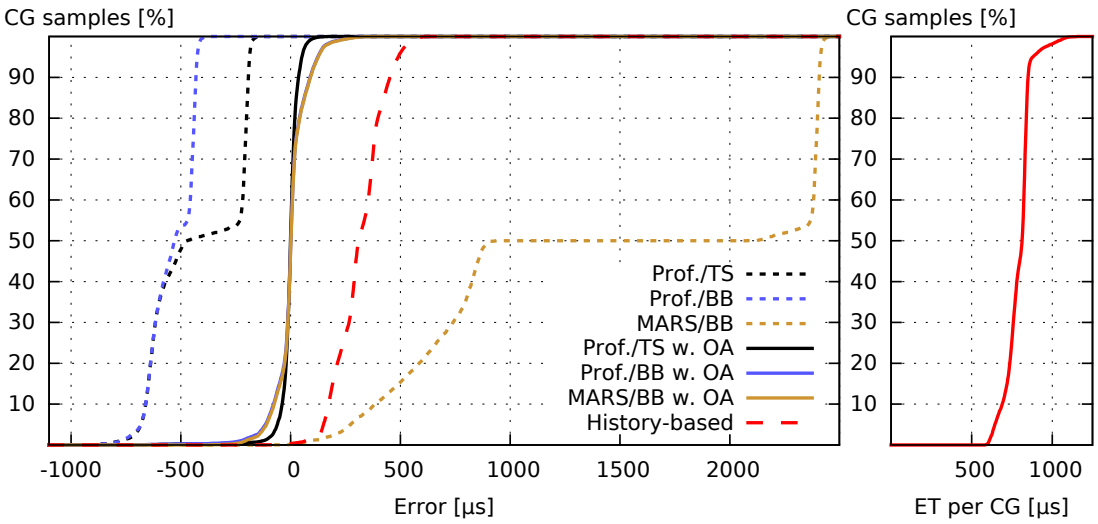


Figure 3.29.: Accuracy of DRAW prediction, speedometer application

CGs drawing the background texture much more than the other ones.

Using online adaption, the TS-based approach shows a lower deviation than the BB-based approaches, since the number of pixels is predicted always very accurately. In contrast, the BB-based approaches predict the CG drawing the pointer texture with changing accuracy. This is because the speedometer application does not use a model view projection matrix and therefore the bounding box created for the vertices changes its size with the angle of the speed-indicating pointer. To this end, online adaption continuously adapts the correction factor for the pointer scene, which slightly reduces accuracy for 50 % of the samples. The history-based approach was able to use its cache for the prediction of only 40 out of the 10 000 CGs. This shows again, that using a history for prediction is not a generic solution at all.

3.8.5.6. Quake 3 "demo four" application

The evaluation results for the Quake 3 "demo four" application are depicted in Fig. 3.30. The majority (more than 93 %) of the CGs had an execution time below 1 ms, the remaining CGs were below 6.1 ms. This distribution explains why the majority of CGs are predicted with small error and a small amount of CGs suffer from errors of several milliseconds. When comparing the BB-based with the TS-based fragment heuristic, the earlier observations (Fig. 3.23) apply again. Namely, the BB-based approach prevents high overestimation of the number of fragments since overlapping areas are not considered, and the TS-based approach prevents underestimation for the opposite reason. Since the sequence of CGs

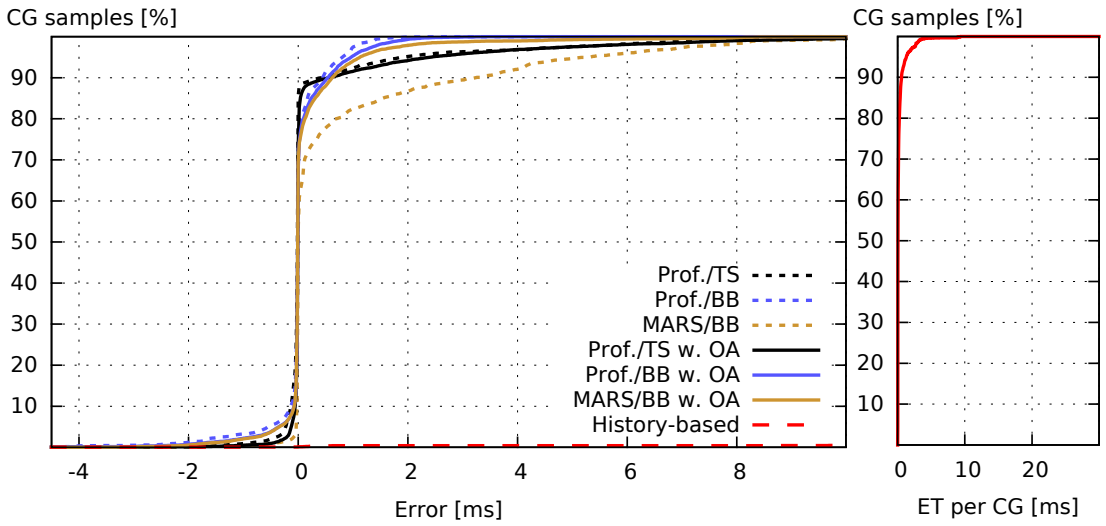


Figure 3.30.: Accuracy of DRAW prediction, quake3 "demo four" application

is extremely heterogeneous—both, huge overlapping parts and sparse areas are drawn with the same shader program—the effect of online adaption is limited. This is due to the fact that Quake 3 uses just two generic OpenGL programs that are used to draw very different parts of the scene, such as background, walls, and players. This implies that for online adaption of drawing very different parts of a scene, the same correction factor is used. In this particular case, the correction factor used most of the time is on average a little greater than 1 for the profiling-based approaches and smaller than 1 for the MARS-based approach. Thus, online adaption shifts the profiling-based approaches slightly towards overestimation, while the MARS-based approach is significantly shifted towards underestimation. Essentially, these evaluation results point out that our concept for online adaption is not effective if CGs very different parts of a scene all use the same correction factor. The TS-based approach again is very effective in preventing underestimation but has a higher overestimation than BB or MARS.

The history-based approach again could use its cache for only 5 CGs and therefore almost always predicted the maximum observed execution time. This lead to an overprediction of at least 24 ms for more than 99.5 % of the CGs[10]. On the other hand, some of the few cache hits resulted in an underestimation since the first of the two identical CGs belonged to a different GPU context. For instance, one CG was predicted with 0.2 ms but took 1.1 ms due to this reason.

[10]This high overestimation was due to a single CG early during the start of Quake 3 that had an execution time of about 30 ms.

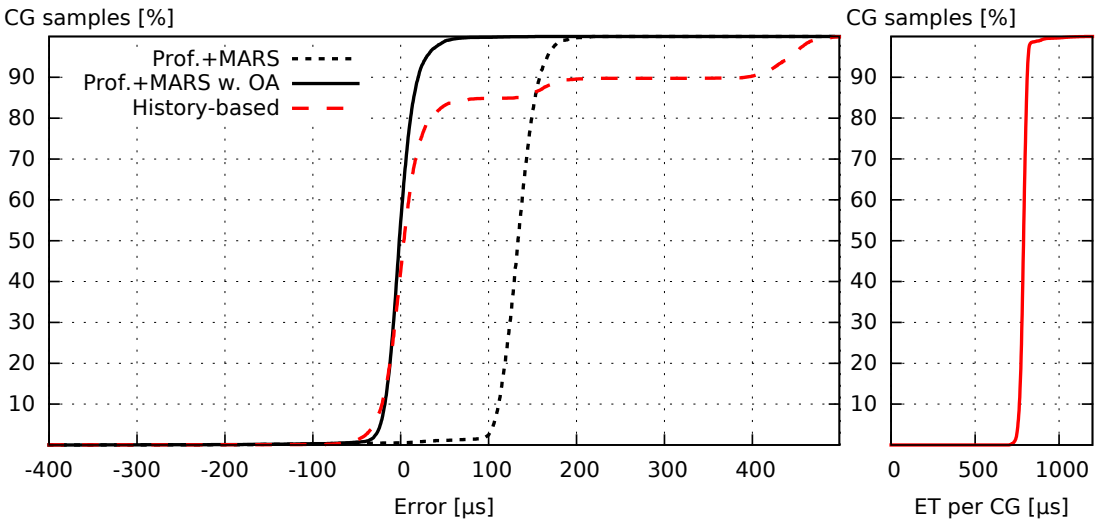


Figure 3.31.: Accuracy of SwapBuffers prediction, glmark2-es2 "build" benchmark

3.8.5.7. SwapBuffers

When an application calls eglSwapBuffers, the respective GPU command is issued in a separate CG. In Fig. 3.31 we depict the prediction error and the absolute execution time of CGs that contain the SwapBuffers operation. The presented results were obtained from the "build" scene of the glmark2-es2 benchmark, other programs show very similar results for our concepts. The execution time of the CGs depends on the size of the framebuffer, and—to a minor degree—on the number of pixels not updated by preceding Draw commands. Thus, the execution time can be predicted quite accurately using profiling. For the profiling-based predictions and the MARS-based predictions, the same analytical model was used (cf. Sec. 3.4.3). The prediction without online adaption overpredicts by about 130 µs on average, which is mainly caused by different loads on the system bus between the different applications and the term $pc_{swapbuffers}$ represents the rounded average over all applications. In our set of evaluated 3D applications, errors of up to about 20 % were observed. If online adaption is used, the prediction is very accurate. The small errors are caused by different loads on the system bus, which can also be observed at the right plot that shows the execution time per CG. Concurrent memory operations affect the execution time of SwapBuffers-CGs, since the GPU shares the main memory and the system bus.

The history-based approach worked—in contrast to Draw-CGs—quite well for SwapBuffers-CGs: almost 84 % of the CGs were predicted using a cached value.

It benefits from the fact that rendering of the "build" scene of glmark2-es2 is very homogeneous. For less homogeneous applications, the history-based approach often fails to use cached execution times. For instance, at the evaluation of Quake 3 its cache hit ratio was below 3 % and none of the cache hits did occur during normal game rendering but only during the animations shown while the game was loading.

3.8.5.8. Impact of fragment heuristics on prediction accuracy

We additionally evaluated the prediction accuracy that would be achieved if the number of fragments would be known precisely. This was achieved with the libETP mode REAL_NUM_FRAG_MODE (cf., 3.7.5), where the prediction is performed after the execution on the GPU and the real number of fragments is available. These evaluations provide a better understanding about how much accurate shader prediction contributes to the accurate prediction of DRAW CGs. The REAL_NUM_FRAG_MODE implies that upon change of the rendering scene libETP injects a "gl_Flush" call that puts different rendering scenes in different CGs. In this section, we evaluated glmark2-es2 benchmarks, since they render only one scene per frame. This implies that the sequence of CGs is equivalent to evaluations without REAL_NUM_FRAG_MODE, which allows to compare the results of this section with the evaluation results in the earlier sections.

In Table 3.8, the mean absolute error (MAE) of the predicted execution times of CGs is depicted. The columns entitled "BB" and "TS" show the MAE from the prediction accuracy results depicted earlier in Fig. 3.26, Fig. 3.27, and Fig. 3.28. The columns entitled "real" show the MAE if the real number of fragments was used. The prediction accuracy results using the libETP mode

Table 3.8.: Influence of the fragment heuristic on the mean absolute error (MAE) of the predicted execution time

		profiling-based			MARS-based	
Application / scene	w. OA	BB	TS	real	BB	real
glmark2-es2 "build"	no	234.4 µs	316.3 µs	294.2 µs	211.0 µs	30.1 µs
glmark2-es2 "shading"	no	788.6 µs	850.1 µs	792.3 µs	110.3 µs	188.4 µs
glmark2-es2 "texture"	no	1452.3 µs	1363.2 µs	1325.6 µs	782.2 µs	2394.6 µs
glmark2-es2 "build"	yes	9.8 µs	15.8 µs	8.0 µs	9.6 µs	7.4 µs
glmark2-es2 "shading"	yes	18.8 µs	38.4 µs	14.5 µs	18.7 µs	13.4 µs
glmark2-es2 "texture"	yes	81.2 µs	79.9 µs	35.5 µs	75.5 µs	33.0 µs

REAL_NUM_FRAG_MODE are depicted in Fig. 3.32, Fig. 3.33, and Fig. 3.34, respectively.

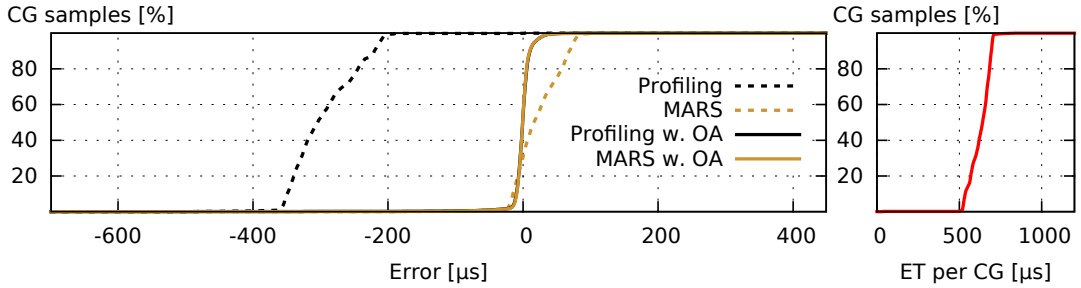


Figure 3.32.: Accuracy of DRAW prediction assuming precise number of fragments, glmark2-es2 "build" benchmark

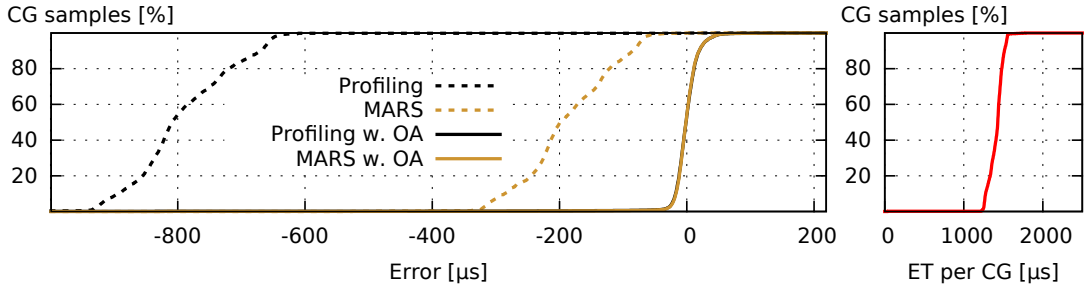


Figure 3.33.: Accuracy of DRAW prediction assuming precise number of fragments, glmark2-es2 "shading" benchmark

Having the exact number of fragments significantly improved the effectiveness of online adaption, since the fragments heuristics introduce jitter by the random selection of triangles (TS approach) and the approximation of rasterization described in Sec. 3.5.1.1. However, having the exact number of fragments does not always improve the overall prediction accuracy. For instance, at the "build" scene, the profiling-based approach has a MAE of 234.4 µs with the BB heuristic, but a MAE of 294.2 µs with the real number of fragments. The profiling-based shader model underestimates the shader execution time, which is slightly compensated by the overestimation of the number of fragments by the BB heuristic (cf., Fig. 3.26 and Table 3.6). Using the real number of fragment removes this compensation and increases—in this particular case—the MAE (cf., Fig. 3.32). Similarly, the number of fragments of the "texture" scene is underestimated by the BB heuristics, which also leads to a higher MAE if the real number of fragments is used (cf., Fig. 3.34). Clearly, the number of

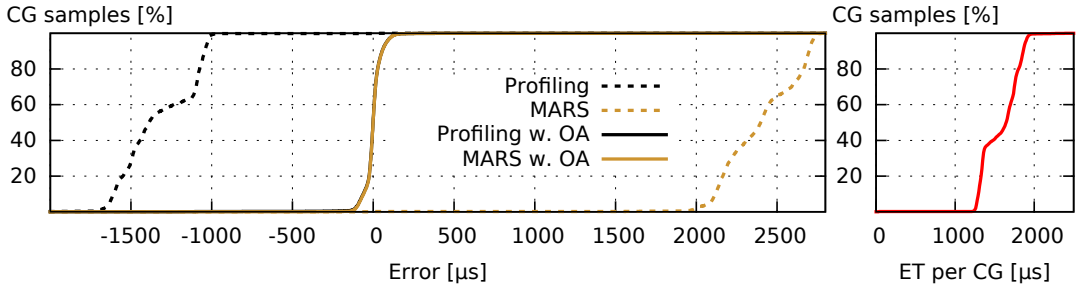


Figure 3.34.: Accuracy of DRAW prediction assuming precise number of fragments, glmark2-es2 "texture" benchmark

fragments has a significant influence on the predicted execution time. However, an error in the shader execution time might often have more impact.

3.8.5.9. Summary of CG prediction accuracy

The previous evaluation results show that our execution time prediction concepts work for existing applications. The prediction of the SWAPBUFFERS command is based on a quite simple linear model that showed errors of up to about 20 %. These errors are mainly caused by different loads on the system bus between the different applications. For the complex DRAW command, we used the submodels for the number of fragment ($m_{\mathrm{nF}}(vertices,\ ctx)$) and the shader execution time ($m_{\mathrm{VP}}(ctx)$ and $m_{\mathrm{FP}}(ctx)$). For $m_{\mathrm{nF}}(vertices,\ ctx)$, we evaluated the TS approach and the BB approach. The TS approach showed only marginal underestimations but can—in some scenes—significantly overestimate the number of fragments. The BB approach cannot prevent underestimations but also is limited concerning overestimation by the size of the render target. In general, if underestimation must be prevented and the application behavior is not known, the TS approach is the better choice. Otherwise, the BB approach is a good alternative. Our evaluations showed that the number of fragments is an important number to achieve accurate prediction but typically the accuracy of the shader execution time has still more impact.

In our evaluations, we compared the profiling-based shader execution time model with the MARS-based shader execution time model. Profiling sometimes significantly underestimated the shader execution time since the used profiling environment did not exactly match the 3D application's environment. In contrast, the underestimation of MARS did not exceed about 15 % and, given that no textures are used, was very accurate. However, significant overestimations of up to about 300 % were observed (at the speedometer

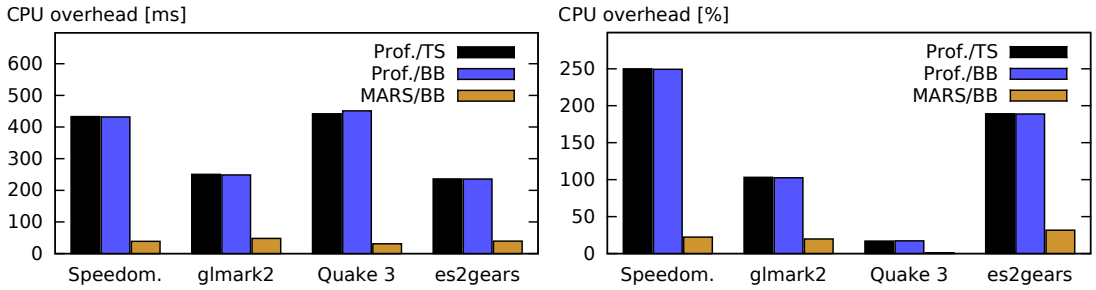
application, Fig. 3.29), since MARS uses a worst-case model to predict the overhead introduced by texture lookups (cf., Sec. 3.5.3.6). For real-time scheduling, preventing underestimations might often be more important than preventing overestimation.

For many scenarios, online adaption could significantly improve accuracy. However, online adaption inherently assumes that the rendered scene does not change fast. Since this assumption is not always valid, online adaption is considered optional and whether it can be used must be carefully decided.

The history-based approach could not benefit from its cache only for a small amount of CGs, since just a small change such as a different model view projection matrix is sufficient to make a CG look different on GPU instruction level. Consequently, this can lead to arbitrarily high overestimations. Additionally, the history-based approach is not aware of the GPU context which occasionally leads to high underestimations if a CG is identical than a cached one but belongs to a different GPU context.

3.8.6. Prediction overhead

The prediction performed by libETP increases the CPU load. Especially on embedded systems, CPU execution time is a valuable resource. Therefore, we measured the CPU time overhead introduced by the execution time prediction of libETP. In order to achieve comparable results, we rendered a defined number of frames for the different approaches, both, with and without libETP being loaded. To this end, new command-line options were added to the speedometer, glmark2, Quake 3, and es2gears applications, which allow to specify the number of frames after which the program terminates. We executed the speedometer application, the "build" benchmark of glmark2, the Quake 3 "demo four", and es2gears with both, exiting after 1 frame and exiting after 1000 frames. All four applications and libETP were compiled with the compiler flag "–O2"—the typical compiler optimization level for high speed. The CPU time was measured with the "time" program available on Linux, which provides the process statistics of the Linux kernel with a granularity of 10 ms. To improve measurement accuracy caused by the limited resolution of the "time" program and side-effects from the Linux system, we used the "time" program on sequences of 100 subsequent runs. From these measurements we calculated the one-time CPU time, which includes the time the program needs to initialize and we additionally calculated the CPU time per rendered frame. In Fig. 3.35, the



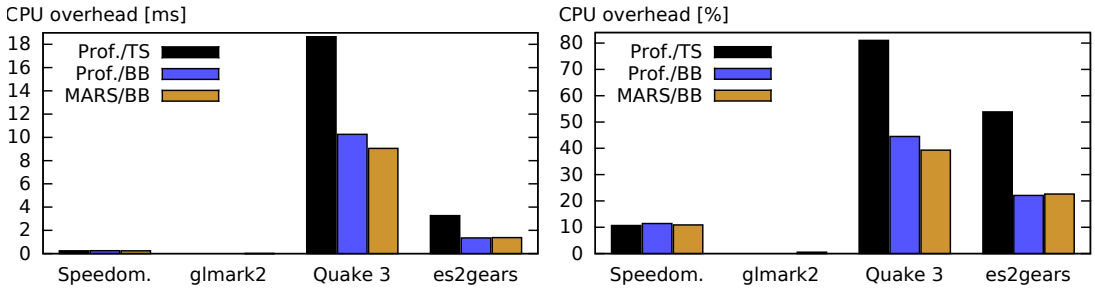
(a) Overhead in ms

(b) overhead in percent

Figure 3.35.: Initial CPU time overhead for loading libETP, compared to native execution

one-time overhead—compared to a native execution without libETP—is depicted. The one-time overhead represents the delay introduced at application startup when using libETP, such as initializing the OpenGL ES Context Monitor, loading the XML configuration file, and mapping the shared memory from the kernel space. The XML configuration file contained already the profiling parameters of the used shader programs. The overhead in time is depicted in Fig. 3.35a. The profiling-based approaches have a significantly higher initial overhead, since the profiling data is stored in an XML file that is parsed initially. For Quake 3 and the speedometer application, this happened twice, since they use two shader programs instead of one, which explains why the profiling-based approaches have a higher initial overhead, there. Our current implementation for loading and saving the XML files use "libxml" and obviously has lots of room for improvement. The MARS-based prediction, which does not use the XML file, delays startup of the application by not more than 50 ms. In Fig. 3.35b, we additionally provide the overhead in relation to the application startup without libETP. Without libETP, the speedometer, glmark2-es2, and es2gears applications start very fast in 173 ms, 243 ms, and 125 ms, respectively. Quake 3 needs 2614 ms before it can start rendering its first frame. This means that the relative overhead for Quake 3 is very low and, when using MARS, it is almost negligible.

The CPU time overhead per frame is depicted in Fig. 3.36. The speedometer application has only few triangles, but since two different shader programs are used, the $m_{\text{draw}}(n_{Calls},\ vertices,\ ctx)$ is used twice per frame. The overhead per frame is only about 250 µs for all three approaches, which is about 11 % of native execution time. The glmark2 benchmark uses a vertex buffer object, which allows to parse the set of vertices (either the bounding box, or the set of

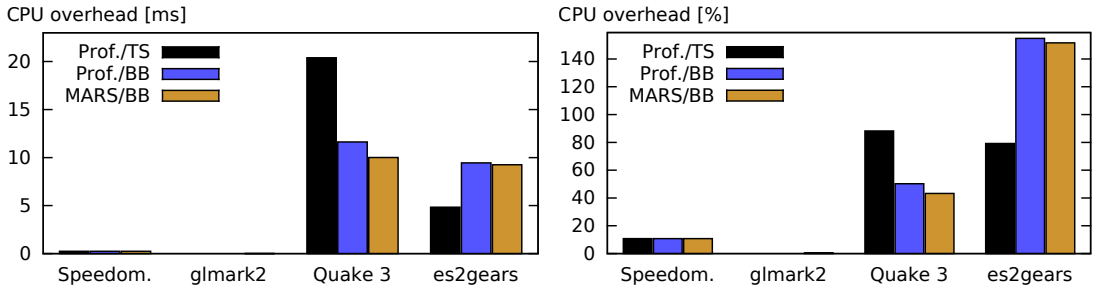


(a) Overhead in ms (b) Overhead in percent

Figure 3.36.: CPU time overhead of libETP prediction per frame, compared to native execution

selected triangles) just once, making prediction extremely fast. The overhead per frame was measured as about 0 s. The Quake 3 "demo four" uses many DRAW calls with different model view projection matrices and does not use VBOs. Thus, the TS approach takes many triangle samples and has the highest overhead. The BB-based approaches have less overhead, since the vertex position calculation must be emulated only for 8 vertices instead of the typically higher number of vertices the TS approach uses. Therefore, the CPU execution time of Quake 3 increases significantly, for the MARS/BB approach by 9 ms, which is about 39 % of the native execution time of Quake 3. Given the complexity of Quake 3's rendering, this is an impressive result. Additionally, rendering 3D objects with many triangles without VBOs—like Quake 3 does—is wasting CPU resources also for a native execution. The es2gears application uses no vertex buffer object, but hundreds of DRAW calls. This requires libETP to calculate the bounding box or the sample triangles again for each frame and is the reason for the overhead of about 1.3 ms to 3.3 ms. Since natively each frame takes about 6.1 ms, the relative overhead is up to 54 %. A scene such as the gears drawn by es2gears would have a much lower CPU overhead if vertex buffer objects were used for both, native and libETP execution.

As explained in Sec. 3.5, our model for DRAW allows predicting multiple OpenGL DRAW API calls in one step to keep the prediction overhead low. More precisely, we collect the vertices OpenGL DRAW API calls and perform the prediction if the rendering scene changes (cf., Sec. 3.3.2.2). Therefore, we additionally evaluated the overhead without this optimization, i.e., with n_{Calls} in the model $m_{\text{draw}}(n_{Calls}, \textit{vertices}, \textit{ctx})$ always being 1. For this evaluation, Line 12 of Listing 3.1 was removed and the Lines 13–14 were also executed "on receive of a Draw(vertexList) call". The results are depicted Fig. 3.37b.



(a) Overhead in ms (b) Overhead in percent

Figure 3.37.: CPU time overhead of libETP prediction per frame, without DRAW optimization

Without DRAW optimization, the speedometer application and the glmark2-es2 benchmark show almost the same results as with DRAW optimization enabled. This is due to the fact that DRAW optimization cannot reduce the number of predictions in these cases. For instance, the glmark2-es2 'build' benchmark uses just a single DRAW call anyway and thus the number of DRAW predictions is the same. The Quake 3 demo benefits from DRAW optimization only slightly, since for many of its DRAW commands also the scene—particularly, the model view projection matrix—changes. The es2gears application benefits most from DRAW optimization, since it uses many DRAW calls and without DRAW optimization 280 DRAW predictions per frame are performed by libETP, but DRAW optimization reduces to only 3 DRAW predictions. The profiling-based TS approach increased from 54 % to 79 %, mainly because with DRAW optimization the TS heuristic could select triangle samples with $p = 0.5$, whereas without DRAW optimization all triangles were used as samples (i.e., $p = 1$). The most significant benefit can be observed at es2gears and the BB-based approach where the overhead per frame increased from about 22 % to about 155 %. The reason is simple: For each of the 280 DRAW commands—each rendering only between 2 and 5 triangles—a much more complex bounding box consisting of 8 vertices and 12 triangles is calculated. These results show that our DRAW optimization concept contributes to a low CPU overhead of our execution time prediction.

The prediction overhead introduced by libETP is quite small and the DRAW optimization can significantly improve performance for applications that use many DRAW calls. A low constant overhead was not the main goal of our implementation and could be reduced significantly, if needed. For automotive scenarios, the overhead per frame is much more important since this overhead is directly related to the complexity of the prediction models. Our concepts for

execution time prediction empowers a GPU Scheduler to provide isolation. To spend a few percent of CPU execution time is reasonable. Moreover, the current implementation of libETP leaves significant room for improvement. For instance, the different modes of operation listed in Sec. 3.7.5 result in many conditional jumps that could be removed for production use-cases.

3.8.7. Evaluation conclusion and summary

The accuracy of the execution time prediction mainly depends on the number of fragments (which is predicted by a heuristic), and the execution time per shader instance.

We evaluated the BB (bonding box) and the TS (triangle samples) fragment heuristics. The BB heuristic is good on average, but heavily depends on the coverage factor, which represents an average of quite different 3D scenes. Clearly, this cannot perfectly match all scenes, nor can it avoid underestimation. The TS heuristic shows only very small underestimations, caused by the random selection of triangles. However, for scenes that include many fragments that do not pass the depth test, it overestimates. This could only be avoided by emulating the depth test on the CPU, which would additionally require to emulate the full vertex processing and rasterization on the CPU, which is not possible in real-time on resource-limited embedded hardware.

For the execution time per shader we evaluated the profiling-based approach and the MARS-based approach. The profiling-based approach runs the shader programs in an emulation environment that is as close to the actual execution as possible. Since the sizes of textures are not known at compile-time of the shader, a default size is used. This default texture might be too small or too big, which has a significant impact on accuracy. To create the MARS models, large sets of training data were used. In our evaluations, we did not underestimate by more than 13 %. However, it occasionally overestimated, since the memory operations, which to some degree run concurrently to the other GPU instructions, were not accurately reflected in the submodels. In contrast to the profiling-based approach, MARS tends to overestimate. The evaluations of the MARS-models show that it is possible to achieve competitive accuracy by using an offline-trained model. This works without profiling during runtime, which potentially affects the scheduling of high-priority applications.

The evaluation results for the execution time per CG showed that the BB heuristic and the profiling-based approach sometimes cause underestimation. On

the other hand, the TS heuristic and the MARS-based approach (the BB heuristic also, but to a minor degree) sometimes cause overestimation. Online adaption significantly reduces the prediction error if the scene changes rarely. However, for programs like Quake 3 that use the same shader for very different scenes, it compensates bias but is much less effective. The overhead introduced by our prediction is relatively small, even on an embedded system with limited CPU resources. For optimized 3D applications such as the glmark2-es2 benchmark, it is even negligible.

The history-based approach proposed by [KLRI11] suffered from rare cache hits, and—even worse—some of the few cache hits belonged to different context, which was not detected by the history-based approach and lead to wrong estimations even if the cache could provide the predicted execution time.

In an automotive context, critical applications such as 3D instruments on the IC are typically very homogeneous, since certifiability is required (cf. Sec. 2.1.6), and according to [ISO03, ISO 17287] the HMI system's interference on the driver must be assessed and limited. For these applications, online adaption is very effective and the prediction error rarely exceeds 100 µs. This prediction error is small compared to the typical *refresh rate* of 60 Hz and therefore can be easily compensated by always using a slightly overpredicted execution time for GPU scheduling (we did so in our evaluations of the GPU scheduling, see Sec. 4.5.1). Non-critical applications such as 3D games played someone else than the driver, might not be very homogeneous. In such cases, using a prediction model that avoids underestimation, such as the combination of TS and MARS, can be a solution.

3.9. Related Work

To enable real-time GPU scheduling, prediction mechanisms for GPU CGs were proposed in several works. Bautin et al. [BDC08, DWA08] designed a system for GPU multi-tasking including a priority-based scheduler, called Graphics Engine Resource Manager (GERM). To this end, GERM collects statistics of the execution time of GPU CGs to calculate an average execution time per CG using a polling mechanism, which is less accurate and needs more CPU resources than our interrupt-driven approach. For each process, they estimate the average execution time per vertex and predict by multiplying with the number of vertices contained in a CG. This prediction model is accurate, if the number of fragments per vertex has low jitter and the used shader programs have similar execution time. Unfortunately, this is not true for the majority of applications. For instance, if each frame of an application (e.g., Glmark2) is rendered using multiple shader programs, their execution time will likely differ. Furthermore, rendered 3D objects often have heterogeneous vertex density (e.g., comparing plain walls of a building with sophisticated vehicle models) or are at different distance from the camera (closer objects typically produce more fragments). In contrast to our approach, their prediction does not consider the OpenGL ES 2.0 Context nor the actual set of GPU commands inside the CGs resulting in inaccurate predictions.

Kato et al. presented another real-time GPU scheduler called TimeGraph [KLRI11]. Our approach borrowed their concept of accurately measuring the execution time of CGs using GPU to CPU interrupts. For prediction, they propose a history-based approach that uses the recorded execution time of previously executed CGs. The prediction algorithm then checks whether a record for the same CG binary code exists. If this is the 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 CGs cannot be predicted. Using the maximum observed execution time in such cases might lead to drastic overestimations 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 [SGDR14]. As shown by the authors, this leads to significant prediction errors for complex dynamic scenes, which occur in many rendering applications today.

In [YZQ+13], 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 rate*. To this end, the computation time of the *Present* command (similar to the eglSwapBuffers command of EGL), which represents the execution of one frame on the GPU, has to be predicted. Similarly to Kato [KLRI11], 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 [KLRI11], since only fully rendered frames are measured and scheduled rather than GPU CGs (typically, a frame is rendered using multiple CGs). Furthermore, they assume that the applications are well known and have a homogeneous rendering behavior. Therefore, an application never calling the *Present* command would be granted infinite GPU execution time, possibly blocking all other applications.

Igehy et al. propose to extend the graphics API to allow for better parallelization [ISH98]. Large rendering tasks are split up to batches of vertices and tiles. Since fragment processing is separated from vertex processing, the number of fragments is inherently given and must no longer be predicted. The concepts used in their "Argus" implementation could thus replace our concepts in Sec. 3.5.1. Unfortunately, Argus uses slow software rendering, since their concept is not supported by existing 3D GPUs. As an alternative to software rendering, [ODK+00] proposes to use custom hardware, i.e., an Imagine stream processor, for rendering. Approaches like parallelized WireGL [BHH00, HEB+01] use the parallel graphics extension from Igehy et al. to use a cluster of multiple nodes for rendering. This approach is improved by Chromium [HHN+08] and AnyGL [YSJZ02]. All the concepts mentioned in this paragraph have in common that they intercept OpenGL API calls, alter them, and stream them to other nodes. Although the focus differs, the idea to intercept OpenGL API calls is also used in this work.

3.10. Summary and future work

3.10.1. Summary

In this chapter, we presented our concepts for the prediction of the execution time of 3D GPU Command Groups (CGs). Our framework is implemented in a shared library called libETP, which intercepts OpenGL ES 2.0 API calls and uses the OpenGL ES 2.0 Context and information from the native GPU driver in user space. Additionally, the Execution Time Monitor in kernel space provides the accurate execution time of CGs after execution.

The DRAW command that is used to perform 3D rendering tasks, depends on the OpenGL ES 2.0 Context, which includes the processing time on the GPU for both, vertex processing and fragment processing. The vertex processing time and the fragment processing time depend on the number of vertices and fragments, and the processing time per vertex and fragment, respectively. To estimate the number of fragments, we presented the triangle samples (TS) approach, which randomly selects samples, emulates the vertex position calculation of the vertex shader, and calculates the covered area. Additionally, we presented the bounding box (BB) approach, which calculates the bounding box of all vertices, emulates the vertex position calculation for the eight vertices of the box, and calculates the size of the covered area. To estimate the vertex shader and fragment shader processing time, we presented a profiling-based approach that executes the shaders in an instrumented benchmarking mode and measures the execution time. Moreover, we presented an alternative approach that uses MARS-based non-linear machine learning models trained for the target system to estimate the shader execution time prior to their first execution. To compensate small prediction errors, an online adaption concept using exponential smoothing was proposed.

Our evaluations show that the bounding box provides a good average accuracy, while the triangle samples approach prevents from significant underestimations but tends to overestimate in some scenes. We also showed that the profiling-based concept for the prediction of vertex and fragment processing works well but that the MARS-based approach can achieve comparable accuracy without interfering with applications that run concurrently to the profiling. For typical 3D scenes, which change rarely, our online adaption significantly improves prediction accuracy and the prediction error rarely exceeds 100 µs. Additionally, the overhead introduced by the

prediction is low for both, application startup and its frame-based 3D rendering. Thus, the achieved prediction accuracy is sufficient for many use cases of real-time 3D GPU scheduling on automotive embedded systems.

3.10.2. Future work

The current implementation of libETP lacks support for a few less important OpenGL functions. For instance, support for glGenerateMipmap should be implemented. It essentially consists of memory operations and its execution time depends on the dimensions of the source buffer. Additionally, the execution time of the vertex processing is not exactly linear, as assumed by our model $m_{\text{draw}}(n_{Calls},\ vertices,\ ctx)$. As shown in [Thi12], vertices seem to be processed in batches, which implies that vertex processing linearly depends on the number of batches. For small numbers of vertices, this can lead to a slightly inaccurate prediction. Therefore, it should be analyzed how the number of vertices per batch can be determined. In principle, the same issue exists with fragment processing, although side-effects from the rasterizer and the tiled alignment of the render target (cf., [Ma14]) makes it more challenging. Additionally, the online adaption concept could improve in some scenarios, if additionally a correction factor for the number of fragments is used.

4. GPU Scheduling

In this chapter, we present our concepts for the scheduling of 3D GPU CGs. The proposed framework uses asynchronous dispatching of CGs, which means that CGs submitted to the kernel space via a system call are not directly dispatched (behavior of typical drivers), but extracted and passed to the scheduler for asynchronous dispatching. The GPU scheduler maintains the set of active 3D processes and its associated scheduling parameters. Given that the execution time prediction is correct, the scheduling policy prevents that a CG is dispatched that can violate a deadline of an application of higher priority. Additionally, the scheduler uses sophisticated mechanisms to achieve a high GPU utilization. Our scheduling concept was implemented and evaluated on an embedded automotive platform. Our evaluation results show that our concepts are effective in fulfilling the requirements for a GPU scheduler described in Sec. 2. Additionally, our concepts feature a very small CPU overhead and a high GPU utilization.

This chapter is structured as follows. In Sec. 4.1 we discuss the requirements. The system model is presented in Sec. 4.2. The scheduling algorithm is depicted and described in Sec. 4.3, followed by a description of the implementation in Sec. 4.4. The evaluation setup and the evaluation results are described in Sec. 4.5. The chapter concludes with an outlook on scheduling with preemptive GPUs in Sec. 4.6, related work in Sec. 4.7, and a summary and future work in Sec. 4.8.

4.1. Requirements

As described earlier (Sec. 2.1), the requirements for rendering in automotive scenarios stem from several sources:

- Direct legal requirements. Example: as regulated by German law (StVZO §57 [Jan11]), the speedometer must be visible and display the current speed.
- Implicit requirements from standards and automotive guidelines like [ISO11, ISO 26262] and [ESO08]. Example: Maximum delays for updates of the screen for applications used while driving.
- OEM-defined requirements. Example: The speedometer shall be rendered stutter-free at 60 FPS.

The requirements imply that a user (e.g., the driver of a vehicle) is not allowed to freely customize the system behavior. The two major aspects of GPU rendering are

1. Guaranteed location and visibility of applications‘ graphical contents – in [GSGH+14,GSGH+15] we have proposed access control mechanisms for safe display sharing.
2. Guaranteed real-time 3D rendering.

Next, we discuss in detail the requirements to guarantee real-time 3D rendering, which necessitates 3D GPU scheduling.

Each 3D application is associated with a *priority*, as presented in Requirement R4.1 (cf., Sec. 2.1.4). The priority is used to guarantee preference to more important or more safety-relevant applications. For instance, in an automotive HMI system, the speedometer could get a high priority, the navigation system a medium priority, and custom third-party applications a low priority.

As pointed out in Sec. 4.2, the GPU driver notifies about each vsync event. A vsync event represents the time when compositing starts. The compositor obviously can only bitblit frames that have finished before compositing starts, i.e., the vsync event. For the applications, this means that vsync events are possible deadlines for 3D rendering, in order to make the content appear on the display in time. We denote the time interval between two consecutive vsync events as the *vsync period*.

According to Requirement R4.2 (cf., Sec. 2.1.4), some automotive applications require rendering with a high *frame rate*, which is represented in the unit frames per second (FPS), and defines the delay between two consecutive frames. For the *frame rate* only values with *vsync period* being an integer multiple of the *frame rate* are allowed. This restriction is because intermediate values only increase computational overhead without providing a better user experience. For instance, for a *refresh rate* of 60 Hz, allowed values for the *frame rate* are 60 FPS, 30 FPS, 20 FPS, and so on. The respective distance between two consecutive frames are one *vsync period*, two *vsync period*s, three *vsync period*s, and so on. In contrast, 45 FPS are not allowed, since the distance between two consecutive frames would be sometimes one *vsync period* and sometimes two *vsync period*s, giving a visual experience to the user that is comparable to only 30 FPS but with a computational effort of 45 FPS. To this end, the desired *frame rate* is a crucial parameter of each 3D application since it determines the amount of required GPU resources.

As motivated in Sec. 1.2.1, the scheduling algorithm needs to support a mixture of 3D applications of different importance. For instance, a 3D speedometer is important and requires a high *frame rate* for stutter-free rendering, while the 3D navigation system is of medium importance and a 3D game is less important. To this end, each application has a *priority*, which represents its importance.

The goals of our scheduling algorithm are to

1. guarantee correct handling of the *priority*, i.e., the requirements of an application with higher priority are more important,
2. guarantee the desired (uniformly distributed) *frame rate*,
3. achieve a high GPU utilization, and
4. be very fast, i.e., the CPU overhead introduced by scheduling shall be small.

The first goal is considered more important than the second, i.e., if the GPU resources are insufficient to fulfill the desired *frame rate* of all applications, higher-priority applications precede lower-priority applications. The third and the forth goals are not hard requirements, but optimization goals.

For our concept, we expect the priorities of applications to be unique. For use-cases where more than one application is considered as most important, unique priorities can be used, nevertheless. If this set of most-important applications is schedulable and the applications' (unique) priorities are higher

than the priorities of all other applications, the scheduling algorithm has to fulfill the desired *frame rate* for each of the most-important application.

4.2. System Model

Before we present our technical contributions, we first introduce our system model and assumptions. Rendering using a GPU is a hierarchical process, where applications commit *command groups* (CGs), i.e., batches of GPU commands, to the GPU. For this purpose, applications typically use a standardized graphics API such as OpenGL or DirectX. This abstraction layer is implemented by the GPU driver, which is partitioned into a set of user space shared libraries, and a kernel space part (e.g., a kernel module). The user space part keeps track of the graphics API's state, compiles shader programs, and creates GPU binary code. The kernel space part initializes the GPU hardware, ensures isolation between different processes, switches between different rendering contexts and performs event handling (e.g., signals a process, that GPU execution has finished). The

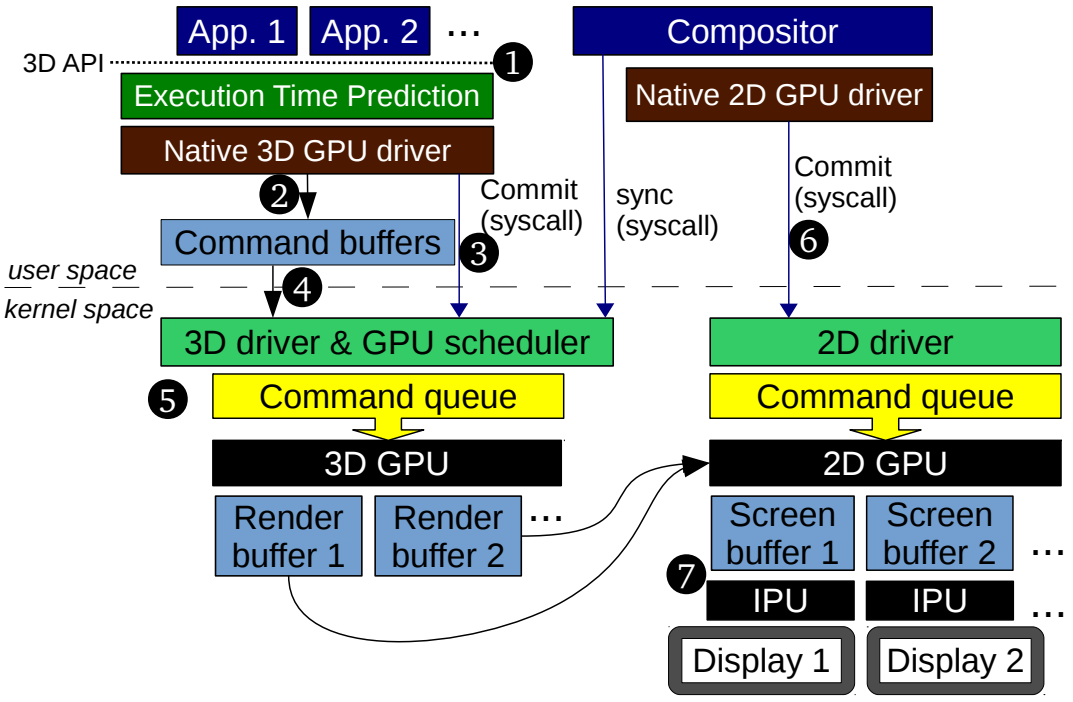


Figure 4.1.: 3D GPU scheduling system model

components and interfaces of our system are depicted in Fig. 4.1. Basically, the system consists of three layers, namely, application-layer, user space driver, and kernel space.

The *graphics application* (e.g., "App. 1") uses the native 3D GPU driver for rendering (❶ in Fig. 4.1). Since the GPU is not preemptive, the GPU scheduler needs to know the execution time of each CG in advance. To this end, the *Execution Time Prediction* predicts the execution time and attaches it to the CG, as described in Chapter 3. From the OpenGL commands, the *native 3D GPU driver* in user space creates a CG in a command buffer (❷) and then notifies the kernel about it (❸). The kernel space 3D GPU driver and scheduler access this data (❹) and place an entry in the 3D GPU's command queue (❺). From there, the GPU fetches the CGs and renders them into the application's dedicated off-screen render buffer.

The *Compositor* is responsible for copying the contents of the off-screen render buffers at the right place into the screen buffer. To achieve this, it waits for a synchronized notification from the GPU scheduler. Modern TFT displays operate at a constant *refresh rate*, typically 60 Hz. The display is connected to a display interface, which streams the screen content (e.g., using HDMI) to the display at this *refresh rate*. If the content of the screen buffer changes while its content is streamed, this effect would be visible to the user as *tearing*. Tearing is a visual artifact, where parts of multiple consecutive frames are stringed together on the display. In order to prevent this unwanted effect, GPU drivers support vertical synchronization (vsync), which allows to update the content of the screen buffer after its content was fully streamed and before streaming starts again, thus avoiding those unwanted artifacts. The GPU driver notifies each time a vsync event occurs, which is used by the GPU scheduler to create compositing tasks (i.e., a set of application windows needing an update). After the compositor receives a task, it uses the API calls of the 2D GPU to create a 2D GPU command batch and to commit it to the 2D kernel space driver (❻), which puts it into the 2D GPU's command queue. From there, the 2D GPU fetches and executes the commands that bitblit (i.e., copy) to the determined place on the screen buffer, which is read by the IPU and displayed on the connected display screens (❼). To allow the 3D GPU to render the next frame while the 2D GPU performs bitblitting, each application uses two alternated buffers (called "double buffering"). Next, we describe our concepts for real-time 3D GPU scheduling in automotive scenarios.

4.3. Approach

In this section, we present our system architecture and scheduling algorithm. Next, we discuss the components of the GPU scheduler and their interaction. Then, we discuss especially the scheduling parameters and the scheduling algorithm.

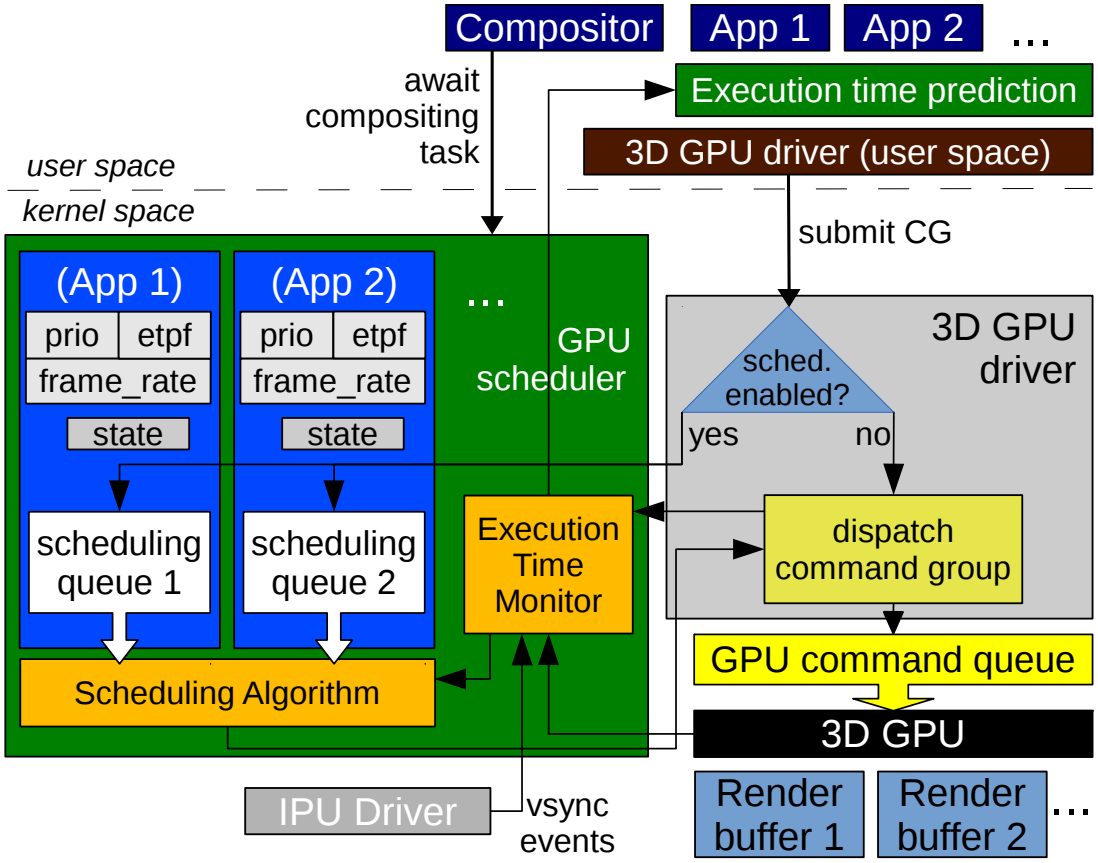


Figure 4.2.: GPU scheduling architecture

4.3.1. System Architecture

Our architecture is presented in Fig. 4.2. As discussed earlier, non-preemptive real-time 3D GPU scheduling requires that the execution times of GPU CGs are known in advance. Our execution time prediction estimates the execution time of each CG in user space in a shared library located between the user space program and the *3D GPU driver* in the user space (cf., Sec. 3.3). The *3D GPU driver* in user space inserts GPU instructions into its command buffer. Once the CG is complete, it uses a system call to submit it to the 3D GPU driver in the kernel space. Without a GPU scheduler, the GPU driver in kernel space

dispatches the CG as part of the system call execution. Dispatching means that the CG and associated data from the user space process are verified and inserted into the GPU's command queue in a specific format understood by the GPU. The GPU then executes the CGs exactly in the order in which they were inserted. Consequently, if the GPU scheduler is disabled, the order in which CGs are submitted to the kernel space GPU driver is the same as the order of execution on the GPU. If the *GPU Scheduler* is enabled (i.e., by loading the GPU scheduler kernel module), it registers with the GPU driver kernel module. This changes the driver behavior such that CGs are no longer directly dispatched but forwarded to the GPU scheduler module instead. For each application, the GPU scheduler maintains a scheduling queue (such as "scheduling queue 1" in Fig. 4.2) for managing the submitted CGs. The *Scheduling Algorithm* uses these scheduling queues, as well as the parameters, such as *priority*, *frame rate*, the timestamps of the vsync events, and the internal state (e.g., the next target deadline) to select the CG that is dispatched next. Eventually, the selected CG is dispatched using the dispatch function of the GPU driver. The Execution Time Monitor keeps track of the relevant timestamps, e.g., when a CG was inserted, when it was dispatched, and when it has finished execution. These timestamps are used by the scheduler to trigger the Scheduling Algorithm to submit new CGs but also for accounting of already consumed GPU resources.

4.3.2. Application-specific parameters for scheduling

As mentioned in Sec. 4.1, each application has a *priority* and a *frame rate*. However, to use only these two parameters for scheduling would imply a negative impact on the GPU utilization. If a low-priority application did submit its CGs faster than a high-priority application, scheduling any of the lower-priority applications now could cause a high-priority CG to get dispatched too late since the dispatched low-priority CG cannot be preempted. Such a delayed submission can be due to unfortunate CPU scheduling or slower application code. Moreover, each time an application enqueues a SwapBuffers CG, it is delayed at least until the next vsync event occurs. Thus, the vsync events are a knowledge horizon, which makes it impossible to estimate what comes beyond. Since low-priority applications often cannot be scheduled, this means that the GPU would inevitably be idle for a significant percentage of time. To exemplify the mentioned disadvantage, we depict in

Fig. 4.3 a possible situation, where both aspects can be observed. The Lines P2 and P1 depict the enqueued CGs, in which estimated execution times are represented by the length of the yellow bars and the beginning of the blue arrow lines represent the time at which the applications submitted the CGs, respectively. In this example, the scheduler first receives the command groups

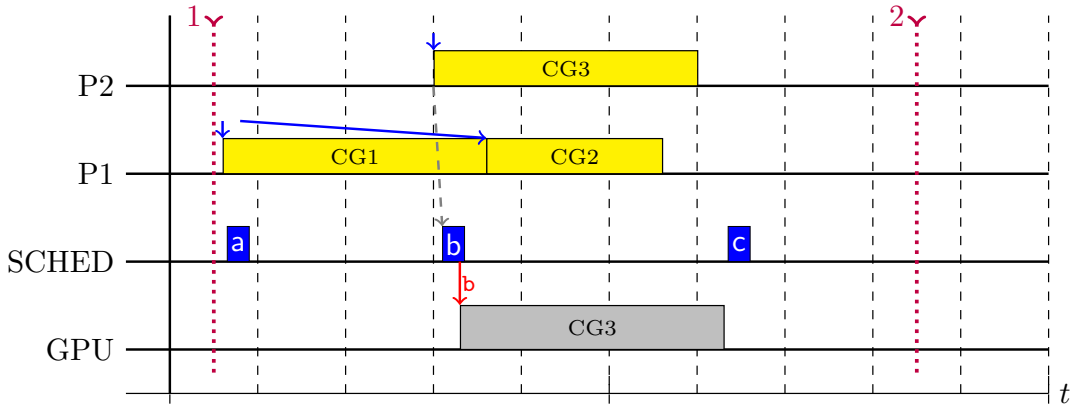


Figure 4.3.: Example for simple priority-based scheduling

CG1 and CG2 from the lower-priority process P1. The Scheduling Algorithm immediately executes "a" and detects that the higher-priority process P2 did not yet finish its frame and did not submit the next CG. In order to eliminate the risk of P2 not meeting its deadline, the scheduler must not schedule CG1 or CG2 from P1. After CG3 from P2 is received, the scheduler immediately dispatches it "b". After this CG has finished execution, the scheduler checks again at "c". However, the CG1 of P1 would not finish before vsync event 2 (red dotted line). In the worst-case, P2 could submit CGs for the next frame directly after vsync event 2 and demand almost all of the available GPU time. Thus, again CG1 must not be scheduled. While strong guarantees to high-priority applications are provided, the GPU is idle most of the time due to the lack of knowledge about the execution times of high-priority CGs not received, yet.

In order to increase the GPU utilization, we introduce the demanded execution time per frame (*etpf*) of an application. The idea is, to tell the scheduler the amount of GPU execution time it is supposed to reserve for each frame of an application. For future frames and while not all CGs of the current frame were received, the scheduler reserves the respective amount of GPU execution time for the application. The execution time per frame (*etpf*) can be determined as part of the software assessment performed by the OEM. To facilitate different rendering scenes of an application, the *etpf* can be updated

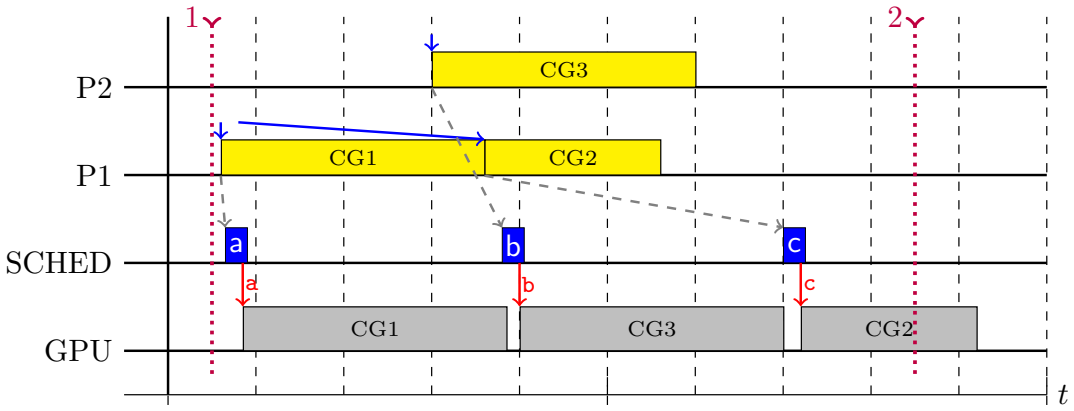


Figure 4.4.: Example for scheduling using execution time per frame (*etpf*)

during runtime using a system call. It might be objected, that the *etpf* is hard to determine for an arbitrary third-party application, where no software assessment takes place. However, in automotive scenarios, third-party applications will have the lowest (or at least a very low) priority assigned. For the lowest-priority application, the *etpf* value is irrelevant, since the *etpf* of an application only affects applications that have lower priority. Additionally, applications can also use $etpf = 0$ if priority-based scheduling of the current frame—without reservations for future frames—is sufficient. In Fig. 4.4, we depict the same sequence of received CGs as in Fig. 4.3 to explain how the scheduler can benefit from an *etpf* parameter. We assume that P2 uses $etpf = 4$. At "a", the scheduler checks, whether CG1 (the only command group available then) can be scheduled, i.e., whether the higher-priority process P2 thereby cannot miss its deadlines. To this end, it checks whether the estimated execution time of CG1 plus the *etpf* of P2 ends before P2's deadline (vsync event 2, red dotted line). Since this is the case, CG1 is dispatched. After CG1 is finished, at "b", CG3 is available and immediately dispatched, since, otherwise, P2 would miss its deadline. After CG3 is finished, the scheduler checks at "c" whether the time when CG2 would finish leaves at least the *etpf* of P2 before vsync event 3 (not depicted). Since this is the case, CG2 can be dispatched. We observe that using *etpf* can drastically reduce GPU idle time.

To summarize, each application has the following scheduling parameters. Each of these parameters can be changed during runtime and immediately affects the applications CGs.

- Unique *priority*

- Desired *frame rate*
- Demanded execution time per frame, called *etpf*

4.3.3. Conceptual Design of the Scheduling Algorithm

In this section, we present the brief conceptual design of our scheduling algorithm. The detailed algorithm is explained later in Sec. 4.3.5. As explained in Sec. 4.3.1, the CGs submitted by the 3D applications are enqueued into dedicated scheduling queues. The scheduling algorithm is executed asynchronously in a dedicated kernel thread. In each run, it selects—if possible—the scheduling queue whose tail CG shall be dispatched next.
The major constraint for all scheduling decisions is

(A) a CG must not be dispatched if this can violate a higher-priority deadline.

Within constraint (A) we try to achieve a high GPU utilization by

(B) scheduling the earliest deadline, since first scheduling a higher-priority later deadline could result in a missed deadline of the lower-priority application having the earlier deadline.

In Listing 4.1 we depict a brief sketch of the scheduling algorithm to show the basic principle. In Line 1, we calculate the point in time at which the CG that is selected and dispatched next by the scheduling algorithm will start executing on the GPU. This point in time is the reference for the subsequent calculations

Listing 4.1: **Scheduling algorithm: sketch how the next CG is selected**

```
 1 Calculate time at which next CG will start executing
 2 candidate = none
 3 For scheduling queue Q from highest to lowest prio:
 4     if Q is not empty:
 5         calculate time at which next dispatched CG must have finished
 6         calculate the available time, i.e., the delta between Line 1 and Line 5
 7         if Q's next CG would fit into the available time:
 8             if the CG's deadline is earlier than the deadline of the existing candidate:
 9                 candidate = Q's next CG
10         else
11             return candidate
12     reserve execution time for Q
13 return none
```

of the algorithm. Next, we iterate through the scheduling queues starting with the one with the highest priority, i.e., the most important one (Line 3). Starting with the scheduling queue of the highest priority, we check whether it contains at least one CG waiting for dispatch, i.e., is not empty (Line 4). If this is the case, we calculate the point in time at which the next dispatched CG must have finished in order to prevent violation of higher priority deadlines (Line 5). In this calculation, we consider reserved execution times of previous iterations of the loop in Line 3.

The available time calculated in Line 6 is the contiguous amount of time from the time the GPU will have finished previously dispatched CGs and the point in time one of the higher-priority scheduling queues needs to start executing on the GPU such that all higher-priority deadlines can be met. In Line 7, we then simply check whether Q's next CG would fit into the available time. If this is the case, and its deadline is earlier than the deadline of the existing candidate (Line 8), this CG is considered as (so far) best candidate for dispatching. If the available time is insufficient and the best candidate (ensured by Line 3 and Line 9) is returned (Line 11). It is possible that in Line 11 "none" is returned. This happens if no CG was found that cannot violate a deadline of a higher-priority scheduling queue and consequently the GPU might become intentionally idle for a short period of time.

In Line 12, the execution time on the GPU required to meet Q's deadlines is imputed as reserved. There, Q's *etpf*, *frame rate*, and—if available—the remaining predicted execution time of its current frame is considered. This calculation is used in Line 5 to determine if dispatching a CG of lower priority could violate deadlines of Q.

4.3.4. Important Parameters, Variables, and Functions

In order to calculate correctly whether deadlines are met, we have to consider not only the execution time on the GPU, but also the CPU execution time of the scheduling algorithm and the GPU driver's dispatch function. The CPU time of the scheduling algorithm and the dispatching by the GPU driver is represented by $SDdelay_C$, which is a conservative estimation, i.e., the upper bound. Additionally, the number of unfinished CGs in the command queue is limited by an upper bound that can be configured by the parameter *MPCG*. For *MPCG*=1, the GPU is idle while the scheduling algorithm or the GPU driver's dispatch function is running. In Fig. 4.5, we show a small example where three CGs of very short execution time

are executed with *MPCG*=1. Since the GPU queue length is one, scheduling and

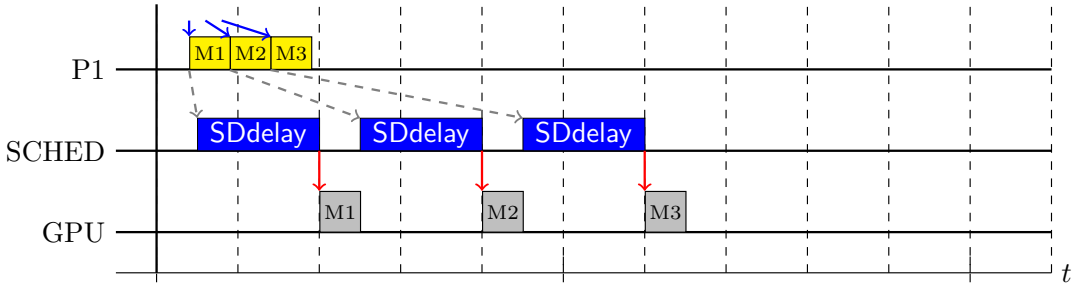


Figure 4.5.: Example for the effect of $SDdelay_C$ using *MPCG*=1

dispatching on the CPU cannot run in parallel to execution of a CG on the GPU. Therefore, the time needed to execute all three CGs, takes their actual execution on the GPU plus three times the delay introduced by scheduling and dispatching, $SDdelay_C$. If *MPCG* is increased to two, $SDdelay_C$ on the CPU overlaps with GPU execution time. Fig. 4.5 shows an example with *MPCG*=2, where two processes submit CGs. Process P1 submits CGs of very short execution time,

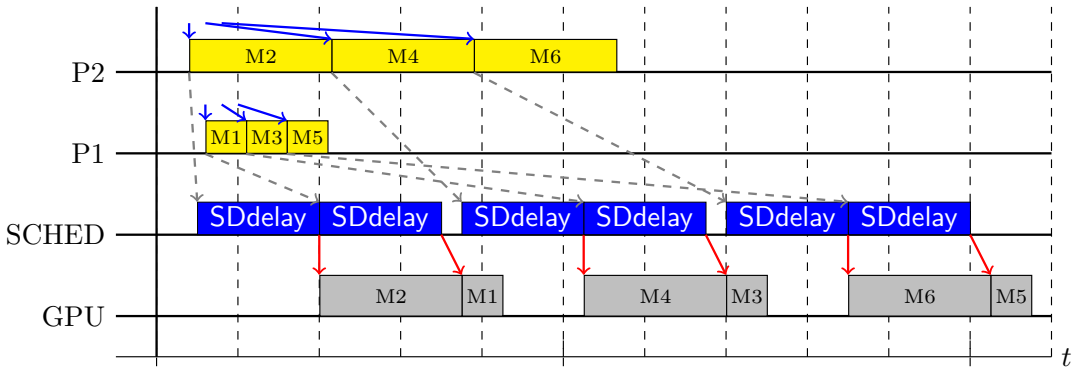


Figure 4.6.: Example for the effect of $SDdelay_C$ using *MPCG*=2

while P2 submits CGs of higher execution time. While having the GPU run in parallel to scheduling and dispatching provides a significant speed-up, the GPU is still idle for some time. We observe that not only CGs of an execution time below $SDdelay_C$ can cause the GPU to be temporarily idle, but also CGs with higher execution times (M2, M4, and M6 in the example) might be delayed. In the example, the start of M4 (and M6) is delayed by $SDdelay_C$ minus the execution times of M1 (or M3). Fortunately, such idleness is very rare in typical scenarios, since the vast majority of CGs takes longer than $SDdelay_C$ to execute. However, to be on the safe side, the worst-case must be considered, i.e., the situation

where some CGs have extremely small execution times. We denote the smallest possible execution time of a CG by et_{MIN}. From the two presented examples, we derive Eq. 4.1, which calculates the worst-case time that has to be reserved for an enqueued CG and additionally define the helper functions Eq. 4.2–4.4.

$$et_{delayed}(predET\ et) = et + max(0, \texttt{SDdelay}_C - (\texttt{MPCG} - 1) \times et_{MIN}) \tag{4.1}$$

$$to\_periods(time\ t) = \frac{t}{\texttt{vsync\_period}} \tag{4.2}$$

$$to\_time(periods\ p) = floor(p \times \texttt{vsync\_period}) \tag{4.3}$$

$$stride(Q) = \frac{\texttt{refresh\_rate}}{Q.\texttt{frame\_rate}} \tag{4.4}$$

The function *to_periods* converts a point in time to the unit "periods". The value is rounded to floor in order to obtain a discrete integer value for the period. The function *to_time* calculates the point in time where a period starts. The function *stride* calculates the number of vsync periods between two consecutive frames based on an application's desired *frame rate*.

Variables representing a sequence number of a specific *vsync period* have "#" as a suffix. The major variables and terms are listed and explained in Math Terms at the end of this work. For each scheduling queue, the scheduling algorithm maintains the following state variables.

reserved GPU execution time that is reserved for the scheduling queue to meet its next deadline.

enqueued Sum of the predicted execution times of the CGs submitted by the application.

dispatched Sum of the predicted execution times of the CGs already dispatched for the next frame.

current# The sequence number of the current *vsync period*.

target# Sequence number of *vsync period* until which the frame shall be completed.

desired# Sequence number of *vsync period* until which the next frame must have finished such that the desired *frame rate* is fulfilled. Used only for statistics and monitoring.

finish# Sequence number of *vsync period* at which the previous frame was actually completed, i.e., the period at which the GPU signaled completion of the CG completing the last frame.

4.3.5. Scheduling Algorithm

Next, we explain the scheduling, which consists of four steps.

1. Submission of CGs by applications (Listing 4.2).
2. Update the scheduling data (Listing 4.3, Lines 1–6).
3. Select the CG with the earliest possible deadline (Listing 4.3, Lines 7–37).
4. Dispatch the selected CG (Listing 4.3, Lines 38–48).

The first step—depicted in Listing 4.2—is executed by an application thread. The *Scheduling Algorithm*—depicted in Listing 4.3—is executed by the scheduler kernel thread and consists of Steps 2 through 4. Each step is explained in the following.

4.3.5.1. Command submission by applications

In Listing 4.2, we depict the code executed when an application submits a new CG. In Line 1, we increment the aggregated execution time by the CG's predicted

Listing 4.2: **submit(CG)**

```
1 CG.Q.enqueued += et_delayed(CG.predET_C)
2 if CG.is_SwapBuffers
3     CG.Q.reserved = CG.Q.enqueued
4     sleep until next incipient arrival time
5 else
6     CG.Q.reserved = max(CG.Q.etpf, CG.Q.enqueued)
```

execution time. The variable *enqueued* holds the aggregated execution time of all CGs submitted for the current frame. If the CG is of type SwapBuffers, the amount of demanded execution time per frame, the *etpf*, is no longer needed and the predicted values of *enqueued* are used (Line 3), and, additionally, the process sleeps until the next incipient arrival time (Line 4). The process will then be woken up, triggered by the interrupt handler of the display driver running each time a vsync event occurs. If the CG is not of type SwapBuffers, at least the *etpf* of the scheduling queue Q is used as *reserved* time (Line 6, cf., Sec. 4.3.2).

Listing 4.3: **schedule_next()** (selects and dispatches next CG)

```
1 for each CG finished in the meanwhile:
2    busyUntil_C += CG.measuredET - CG.predET_C
3    busyUntil_O += CG.measuredET - CG.predET_O
4    if CG.is_SwapBuffers
5       CG.Q.target# = max(CG.Q.desired#, CG.Q.finish#) + stride(CG.Q)
6       CG.Q.desired# = CG.Q.target#
7 tRef_C = max(busyUntil_C, NOW + SDdelay_C)
8 tRef_O = max(busyUntil_O, NOW + SDdelay_O)
9 max# = current# + FPLA
10 res[] = res2[] = Qdis = 0
11 for priority p = MAX TO MIN:
12    Q[p].target# = max(Q[p].target#, to_periods(tRef_O))
13    if (Q[p] not empty):
14       earliest# = to_periods(tRef_O + Q[p].nextCG.predET_O)
15       Q[p].target# = max(Q[p].target#, min(earliest#, max#))
16       sum_periods = accounted_time = 0
17       for i = (max#) TO current# STEP -1:
18          accounted_time = max(accounted_time, sum_periods)
19          sum_periods += vsync_period
20          accounted_time += res[i]
21          if (res[i] > 0 OR res2[i + 1] > 0:
22             accounted_time = min(accounted_time, sum_periods)
23          accounted_time += res2[i]
24       accounted_time += tRef_C
25       if (Q[p].nextCG.predET_C + accounted_time <= sum_periods):
26          if (Qdis = 0 OR Q[p].target# < Qdis.target#):
27             Qdis = Q[p]
28       else if (accounted_time > sum_periods):
29          break loop
30    if (stride(Q[p]) = 1):
31       res[Q[p].target#] += Q[p].reserved - Q[p].dispatched
32       for i = (Q[p].target# + 1) TO (max#):
33          res[i] += Q[p].etpf
34    else:
35       res2[Q[p].target#] += Q[p].reserved - Q[p].dispatched
36       for i = Q[p].target#+stride(Q[p]) TO max# STEP stride(Q[p]):
37          res2[i] += Q[p].etpf
38 if Qdis > 0:
39    if Qdis.nextCG.is_SwapBuffers:
40       Qdis.dispatched = 0
41       Qdis.reserved = Qdis.etpf
42       Qdis.target# += refresh_rate / Qdis.frame_rate
43    else:
44       Qdis.dispatched += Q[p].nextCG.predET_C
45    dispatch(Qdis.nextCG)
46    busyUntil_C = max(busyUntil_C, NOW) + Qdis.nextCG.predET_C
47    busyUntil_O = max(busyUntil_O, NOW) + Qdis.nextCG.predET_O
48    dequeue(Qdis.nextCG)
```

4.3.5.2. Update the scheduling data

First, the scheduling algorithm (Listing 4.3) updates scheduling times using the finish times of CGs that have finished since the previous run of the scheduler (Line 1). The variable *busyUntil* holds the timestamp at which the GPU has finished (or will finish) all previously dispatched CGs, either using the conservative $busyUntil_C$ or the optimistic $busyUntil_O$ predicted execution times. The purpose of having a conservative and an optimistic estimation is discussed in Sec. 4.3.6.4. Using the measured execution time of the CG, we correct a possible prediction error in *busyUntil* (Lines 2-3). If the finished CG was of type SwapBuffers, which means that the respective application just completed a frame, its next target deadline and desired deadline are updated for the application's next frame using the desired *frame rate*. If the application did not meet its desired deadline, its next deadline is deferred, using *finish#*. Similarly, if the frame was completed a *vsync period* earlier than required, the next deadline is set using *desired#* (Lines 5-6). The desired deadline (*desired#*, Line 6) is used for monitoring and evaluating the correctness of the algorithm, since *desired#*<*finish#* implies a missed deadline.

4.3.5.3. Select the CG with the earliest possible deadline

Our scheduling algorithm guarantees—provided that the prediction is correct—that it will never dispatch a CG that could violate a deadline of any other application that has higher priority. In order to achieve a high GPU utilization, it selects the earliest possible deadline that can be scheduled safely, i.e., without violating higher-priority deadlines. The two *tRef* variables (Lines 7–8) hold the timestamp at which the next selected CG starts executing, either based on *busyUntil*, or—if the GPU is idle—based on the current time. The delay of the scheduling algorithm itself plus the delay introduced by dispatching are denoted by *SDdelay* and postpone *tRef*. In order to limit the number of vsync periods we have to look into the future, we use the scheduling parameter *FPLA* as the maximum number of periods we have to look into the future to be sure that no higher-priority deadlines are violated. Depending on the allowed *frame rate*s, the *FPLA* must be set properly, as explained in Sec 4.3.6.4. The variable *max#*, which is based on *FPLA*, holds the maximum period we have to check (Line 9).

In Line 11, we iterate in the main loop through all available scheduling queues, ordered from highest to lowest priority. At the beginning of the loop, we

ensure that the target deadline is not set before the next CG can start executing (Line 12), since this would result in unjustified reservations that could prevent lower-priority queues from being dispatched. If a scheduling queue *Q[p]* is not empty (Line 13), it might be a candidate for dispatching. In this case, the predicted execution time of the next CG is available and the earliest possible next deadline for this queue is the finish time of this CG, if it would be dispatched next (Lines 14–15). In the Lines 16–23, the scheduling algorithm calculates the accounted time, i.e., the amount of time that is reserved from the future periods. The reservation concept is presented in detail in Sec. 4.3.6.2. Besides the reservations, also the time tRef_C is not available for dispatching and therefore added to *accounted_time* (Line 24). If *sum_periods* is greater than the *accounted_time*, the difference is available time that could be occupied by the application's next CG, which is therefore a candidate (saved in *Qdis*, Line 27). Additionally, Line 26 ensures that the earliest possible deadline is selected. Otherwise, if there is no available time at all, no lower-priority CG can be dispatched and therefore we leave the main loop (Lines 28–29).

In the Lines 30–37, we merge the execution times of the current scheduling queue *Q[p]* into *res[]* and *res2[]* to prevent that scheduling one of the lower-priority queues checked next can result in a deadline miss of *Q[p]*.

The *stride* denotes the number of vsync periods between the deadlines of two consecutive frames of the application. If $stride = 1$, the reservations are added to *res[]*, whereas if $stride >= 2$, the reservations are added to *res2[]*.

For the application's current frame we reserve Q's field *reserved* minus Q's field *dispatched* (Line 31 and Line 35). If the application did not yet enqueue a SwapBuffers CG, at least its *etpf* is reserved (cf., Listing 4.2, Line 6). The variable *dispatched* holds the aggregated execution time of CGs already dispatched for the current frame. For all future frames of the application, its *etpf* is used with the *stride* as step size (Lines 32–33 and Lines 36–37).

4.3.5.4. Dispatch the selected CG

Finally, in the Lines 38 to 48, we dispatch the selected CG. If the CG is of type SwapBuffers (Line 39), we additionally reset the *reserved* and *dispatched* execution times (Lines 40+41) and increment the target deadline. Thus, the next time the algorithm executes, the *etpf* of *Qdis* is correctly accounted for the next target deadline. Moreover, the *target#* is updated to the queue's next deadline (Line 42). If the CG is not of type SwapBuffers (Line 44), this means the

frame is not yet fully dispatched and the variable *dispatched* is incremented by the execution time of the selected CG. In Line 45 the CG is actually dispatched using the native 3D GPU driver. The GPU driver provides us the exact time at which the CG is submitted to the GPU. This time (denoted as *NOW*) is used to update the *busyUntil* variables to the expected finish time of the dispatched CG (Lines 46-47). Eventually, the CG is dequeued from the scheduling queue (Line 48).

4.3.6. Reservation Concept and Schedulability

In this section, we briefly discuss correctness and schedulability tests of our approach. As motivated in Sec. 1.2.1 and Sec. 1.2.2, our scheduler needs to be non-preemptive, isolate higher from lower priorities, consider the desired *frame rate*s, and support dynamic task sets.

By implication, the scheduler must be idling, since non-idling scheduling approaches could also dispatch a long-running low-priority CG that would cause a high-priority CG to miss its deadline. Moreover, a high GPU utilization, as well as a low CPU overhead are desirable in order to achieve low hardware cost, installation space, and energy consumption. Next, we discuss the trade-off between GPU utilization and CPU overhead, and the concept used in our scheduling algorithm.

4.3.6.1. Execution time efficiency on GPU and CPU

A schedulability test for idling, non-preemptive scheduling, is known to be NP-complete [GJ79] (Annex 5). Since the task set is dynamic, this calculation would have to happen in real-time on the embedded platform, which is not feasible.

Our scheduling approach takes advantage of the specific use-case of GPU scheduling to fulfill the requirements. The deadlines are aligned to the vsync events, since the rendered content is supposed to be shown on a display, which is operated with the *refresh rate*. Additionally, the arrival time of a CG cannot be earlier than two *vsync period*s before its deadline. Thus, we do not only ensure that the rendered content is very recent, but also the complexity of scheduling decisions is reduced. The earliest possible arrival time of a CG (i.e., two *vsync period*s before its desired next deadline), is called incipient arrival time. Moreover, the desired *frame rate* of a 3D application is typically not below 20 FPS, since showing only static images would not require 3D rendering

and low *frame rate*s would be perceived as stuttering or flickering[1]. This is beneficial, since lower *frame rate*s would require a further look into the future to make sure all deadlines are met.

4.3.6.2. GPU execution time reservation concept

Our scheduling algorithm uses two arrays to maintain the reserved GPU execution times for future *vsync period*s. The array *res[]* holds the reserved time of scheduling queues that have a *frame rate* equal to the *refresh rate*. This implies that the CGs arrive (typically very early) in the *vsync period* whose end is also their deadline. The array *res2[]* holds the reserved time of scheduling queues that have a *frame rate* lower than the *refresh rate*. This means that the application can submit CGs already one *vsync period* earlier.

Fig. 4.7 shows an example how reservations of higher-priorities are considered in the scheduling algorithm. The example uses $FPLA$=4, which is sufficient to illustrate the concept. The example shows a situation where the main loop of the scheduling algorithm (Line 11) has been executed a few times and the array *res[]* and the array *res2[]* contain the reserved times of the previous loop iterations, i.e., the reservation times of scheduling queues with higher priority. For instance, *res*[*current#*] contains the amount of GPU execution time that has

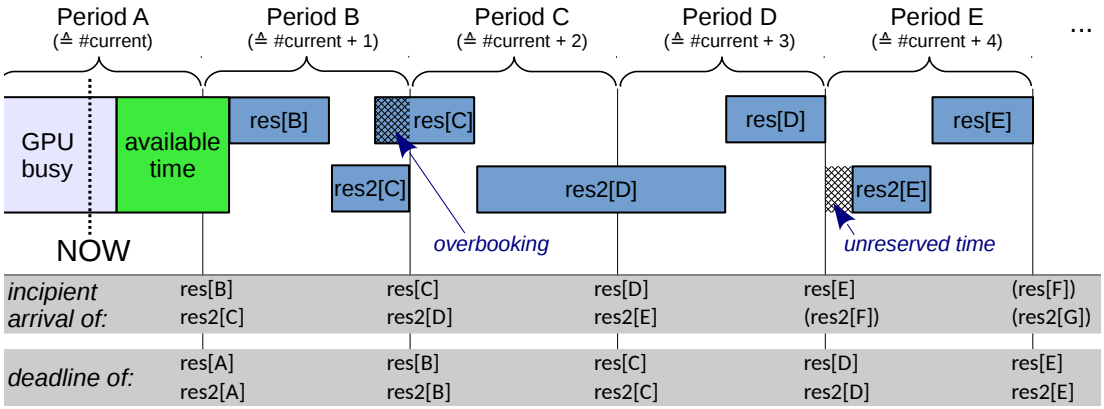


Figure 4.7.: GPU scheduling algorithm reservation example

an incipient arrival time at the beginning of Period *current#* and the end of Period *current#* as deadline. Incipient arrival means, that the application can start submitting CGs from that point in time on. Depending on CPU scheduling and the application's required CPU resources, the actual arrival time can be

[1] Above 15 FPS the vision center in the brain gives the sensation of visual continuity, [RM00], Page 24

slightly delayed[2]. The amount of GPU execution time that has an incipient arrival time at the beginning of Period B and the end of Period C as deadline is held in *res2*[C]. This implies that the reservations held in *res2[]* span two periods, while the reservations held in *res[]* span only one period.

When the scheduling algorithm is executed at time *NOW*, it first determines for how long the GPU is busy with previously dispatched CGs, since this represents the start of the available time. The end of the available time is determined as follows. The algorithm starts with the Period *current#* + *FPLA* (E in this example). It assumes the GPU finishes the reservations just in time. Since the CGs for *res*[E] cannot be dispatched before the beginning of Period E (since they do not arrive earlier), we can without loss of generality assume that those CGs are all executed directly before the end of Period E. The CGs for *res2*[E] could then be executed before. In this case, *res*[E] + *res2*[E] $<$ *vsync period*. Since earlier reservations such as *res*[D] and *res2*[D] have to finish before the beginning of Period E, the Period E cannot fully utilize the GPU, but leave a small amount of unreserved time. If a *vsync period* contains unreserved time, this means, that—if only the reserved amount of GPU execution time of scheduling queues with higher priority would be dispatched—the GPU would inevitably be idle. The case of unreserved time is covered by the scheduling algorithm (cf., Listing 4.3) in Line 18.

The algorithm continues with checking Period D. Again, *res*[D] is placed after *res2*[D], since *res*[D] arrives not before the beginning of Period D, while the incipient arrival time of *res2*[D] is already at the beginning of Period C. This allows to fulfill *res2*[D], although *res2*[D] $>$ *vsync period*.

Looking at the Periods C and D, it can be observed that *res*[C] + *res2*[D] + *res*[D] $>$ 2 $\times$ *vsync period*. Since these reservations do not arrive before the beginning of Period C, it is impossible to meet all existing reservations. To this end, the apparent overbooking can be safely ignored, which is implemented in Line 21 and Line 22 of the algorithm.

The scheduling algorithm additionally prefers earlier deadlines (Line 26), and—if CGs have the same deadline—it prefers the one with the highest priority. This means, that tasks with *target#* = *current#* are preferred over tasks with *target#* = *current#* + 1, cf., the execution policy described in Sec. 4.3.6.1. For instance, at the beginning of Period C, the CGs of *res*[C] and *res2*[D] arrive, but the CGs represented by *res*[C] have an earlier deadline than

[2]Applications such as glmark2-es2 or our Speedometer application achieved a very small submission delay of less than 150 µs (less than 1 % of the *vsync period*)

the CGs represented by *res2*[D]. Therefore, fulfilling both, *res*[C] and *res2*[D], is only possible if *res*[C] is preferred over *res2*[D]. For instance, this ensures that task sets that consist only of applications a with $stride(a) \in \{1, 2\}$ and do not require more than 100 % of GPU execution time on average, are schedulable (see next section). Our reservation concept allows us to calculate the time span between busyUntil and the latest possible start of the reservations, which is the available time.

4.3.6.3. Schedulability

In this section, we discuss how the schedulability of task set can be determined and what GPU utilization is achievable. This allows a system integrator to judge whether the requirements of a given set of important 3D applications can all be fulfilled, based on the application assessment at which the *etpf* values were determined. Our concepts presented in Sec. 4.3.6.1 reduce the complexity of scheduling decisions. Let Q be the set of all scheduling queues and $\forall q \in Q : q.stride = \frac{refresh\ rate}{q.frame\ rate}$. According to the definition of the *frame rate* in Sec. 4.1, all *stride* attributes are integers of the value 1 or greater. Our scheduling algorithm uses two types of reservations:

1. command queues q with $q.stride = 1$ (array *res[]*)
2. command queues q with $q.stride > 1$ where the execution on the GPU can span over two *vsync period*s (array *res2[]*), e.g., *res2*[D] in Fig. 4.7.

In Sec. 4.3.6.2, we showed how the reservations can be put in a sequence. We further showed how this sequence is correctly aligned to the *vsync period*s such that—if possible—no deadlines are missed.

This insight can be used to understand how it can be verified whether a given set of tasks is guaranteed to be schedulable, i.e., meet all deadlines. Since the exact execution times of the CGs are not known in advance, the *etpf* and *frame rate* values are used. In this section, the term "task set" denotes the set of 3D applications that are all considered important and therefore shall get their desired *frame rate*s fulfilled. Optionally, further 3D applications with lower priorities can use the remaining GPU resources, but might not get their desired *frame rate*s fulfilled.

Let T be a task set. The reservations of tasks $t \in T$ with $t.stride = 1$ are all equal, i.e., $\forall p > 1 : res[1] = res[p]$. If all tasks t of the task set have $t.stride = 1$, the schedulability test is trivial: T is schedulable $\Leftrightarrow \sum_{t \in T} t.etpf \leq$ *vsync period*.

Fig. 4.8 shows three examples of possible reservations and how their CGs could be dispatched on the GPU. Example a) shows the case where all tasks t have $t.stride = 1$. To test schedulability, only one period has to be checked, which is also obvious from the presented formula.

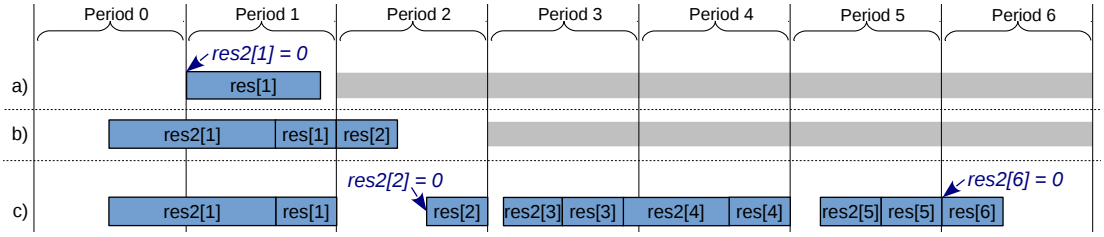


Figure 4.8.: GPU scheduling algorithm schedulability example

Next, we look into situations that contain at least one task t with $t.stride > 1$, which applies to Example b) and Example c). Without loss of generality, we assume that all tasks of the task set have their first deadline at Period 1. This is actually the worst case since the reservations for tasks t with $t.stride >= 3$ will be distributed most non-uniformly. The reservations of tasks t with $t.stride > 1$ have their first deadline at Period 1 and their next deadline thereafter not before Period 3, which implies $res2[2] = 0$.

If all tasks t have either $t.stride = 1$ or $t.stride = 2$ (cf., Example b) in Fig. 4.8), the schedulability test is also relatively easy. In Period 1, $res[1]$ can be dispatched at the end. Thereafter, $res[2]$ can be dispatched at the beginning of Period 2. Before $res[1]$, $res2[1]$ can be scheduled. If $res2[1]$ does not extend more into Period 0 than what is left of Period 2 after $res[2]$, the task set is schedulable, since in Period 3 the reservations continue as in Period 1 in a periodic fashion. Thus, only two periods have to be checked to decide on schedulability: Let $T[i] = \{t \in T | t.stride = i\}$. T is schedulable $\Leftrightarrow 2 \times \sum_{t \in T[1]} t.etpf + \sum_{t \in T[2]} t.etpf < 2 \times vsync\ period$.

In the general case of tasks with different *stride*, multiple periods have to be checked. In Period 1, all tasks have a deadline. After Period 1, their deadlines are given by their respective *stride* as intervals. Eventually, in a future period, all tasks again have a deadline in the same period. This repeats periodically each least common multiple (LCM) of the set of stride values. Formally, $etpf\_lcm = LCM(\{n \in \mathbb{N} | \exists t \in T : stride(t) = n\})$.
Furthermore, according to the description in Sec. 4.3.6.2, $res[i] = \sum_{t \in T[1]} t.etpf$ and $res2[i] = \sum_{t \in T \setminus T[1]} \begin{cases} t.etpf, \text{if } stride(t) \bmod i = 0 \\ 0, otherwise \end{cases}$.

Listing 4.4: **is_schedulable(task set)**

```
1 sum_periods = accounted_time = 0
2 for i = etpf_lcm TO 1 STEP -1:
3    accounted = max(accounted_time, sum_periods)
4    sum_periods += vsync_period
5    accounted_time += res[i]
6    if (accounted_time > sum_periods):
7        return NO
8    accounted_time += res2[i]
9 return accounted_time - sum_periods + res[etpf_lcm] < vsync_period
```

In Listing 4.4, the schedulability test is provided in pseudo code. It follows the GPU execution time reservation concept explained in Sec. 4.3.6.2. The for loop (Lines 1–8) matches closely the corresponding Lines 16–23 of the scheduling algorithm (cf., Listing 4.3). Only the Lines 22–23 of the scheduling algorithm deviate from the Lines 6–7 of the schedulability test. The reason is, that the scheduling algorithm ignores overbooking that cannot be affected by the next scheduling decision, while the schedulability test detects overbooking and returns "NO", indicating that this task set is not schedulable. In Line 9, the schedulability test checks whether the amount of accounted time that extends into Period 0 fits into the amount of time left in Period $etpf\_lcm$. If this is the case, the same sequence of reservations can be repeated for unlimited times and the task set is schedulable for any period of time. The amount of time left in Period $etpf\_lcm$ is simply *vsync period* - $res[etpf\_lcm]$, since $res2[etpf\_lcm] = 0$. In Period 1 all tasks have a deadline and—due to the definition of LCM—in Period $etpf\_lcm+1$ again all tasks have a deadline. If $res2[etpf\_lcm] > 0$, it could only come from an application a with $a.stride = 1$, which, however, contradicts the definition of *res2[]*.

The Example c) in Fig. 4.8 shows a schedulable sequence of reservations for tasks t having $t.stride \in \{1, 2, 3\}$. Therefore, $etpf\_lcm = LCM(1, 2, 3) = 6$ and the schedulability test looks into Periods 1 through 6. Tasks $\in T\,[1]$ have deadlines in each period. Tasks $\in T\,[2]$ have deadlines in Period 1, 3, and 5. Tasks $\in T\,[3]$ have deadlines in Period 1 and 4. Therefore, the highest amount of reserved GPU execution time in *res2[]* is in Period 1 and $res2[1] = res2[3] + res2[4]$. Unreserved time is observed in the Periods 2, 3, and 5. Reservation $res[6]$ is shown at the beginning of Period 6 to indicate that the (periodic) execution can continue with $res2[7]$ (which is equal to $res2[1]$).

For each task set whose tasks t have $t.stride \in \{1, 2\}$, our scheduling algorithm can account for 100 % of the GPU resources. This means, that a task set that

does not de facto require more than 100 % of the GPU execution time, is always schedulable. For task sets that contain tasks t with $t.stride > 2$, this is not always the case, since unreserved times cannot be always avoided (cf., Example c) in Fig. 4.8). As explained in Sec. 4.3.6.1, typical automotive 3D rendering task sets do not contain a task t with $t.stride > 3$. In this case, our scheduling algorithm can account for at least 67 %, which happens if the GPU is idle every third period. To improve this, tasks could be started such that their first deadline is not at the same period. To this end, even with a task set T, with $\forall t \in T : t.stride \in \{1, 2, 3\}$, the scheduling algorithm can often account for close to 100 %.

4.3.6.4. Coping with errors of execution time prediction

The effectiveness of the scheduling algorithm (cf., Listing 4.4) depends on accurate execution time prediction. While our concepts for execution time prediction (cf., Sec. 3) provide good results, they cannot always avoid prediction errors. Additionally, the *etpf* is used to reserve GPU execution time before the prediction actually takes place. This implies that the actual and also the predicted GPU execution time is typically lower than the *etpf*. Both aspects were considered in the scheduling algorithm and are explained next. From the predicted execution time received from libETP, we derive the conservative prediction $predET_C$, which adds a small safety margin, and the optimistic prediction $predET_O$, which subtracts a small margin. In the scheduling algorithm, we calculate $busyUntil_C$ and $busyUntil_O$ (Lines 2–3 and 46–47), which represent the point in time when the GPU becomes idle, based on conservative and optimistic prediction. Similarly, we calculate $tRef_C$ and $tRef_O$ (Lines 7–8), which represents the time the next CG can start executing on the GPU; again, based on conservative and optimistic prediction.

Underestimation of CGs potentially leads to deadline misses of higher-priority applications, since the GPU stays busy longer than expected. To reduce the probability of underestimation, all reservations are based on conservative execution time prediction.

In Line 12, the scheduling algorithm postpones the next deadline if it is impossible to be met. This avoids considering impossible reservations, thus occasionally increasing GPU utilization. However, the decision to postpone a deadline implies that the particular deadline is missed. To avoid that this happens with low confidence, postponing deadlines is based on optimistic prediction, i.e., the deadline is only postponed if—even under optimistic

assumptions (i.e., using $tRef_O$)—the deadline cannot be met.

As explained in Sec. 4.3.6.3, reservations of a sequence of $etpf\_lcm$ periods have to be checked to determine if a task set is schedulable. For the scheduling algorithm, it is important to check the same number of periods to determine the available time (cf., Sec. 4.3.6.2). However, if an application has already submitted CGs for its current frame, the scheduling algorithm uses their predicted execution times as reserved time. Since applications can submit CGs up to two periods before their next deadline, the scheduling algorithm has to check a total of $FPLA = etpf\_lcm + 2$ periods, starting with the current one. Thus, the range from Period *current#* + 2 to Period *current#* + etpf_lcm + 1 contains only *etpf*-based reservations, which guarantees that the *etpf* values are reserved.

Using a command queue length of 1 ($MPCG$=1), the scheduler precisely knows when the CG dispatched next will start executing. If a CG was significantly underestimated, in the worst-case, a higher-priority deadline is missed. To reduce the probability that this can happen, the scheduling algorithm prefers in general CGs with higher priority, since the main loop in Line 11 starts with the highest priority and the first non-empty queue found is assigned to Qdis (Line 26). Only if a lower-priority application has an earlier deadline, it can be scheduled earlier. If $MPCG$>1, in rare cases, multiple of the CGs currently pending in the command queue might be significantly underestimated. This increases the probability that higher-priority deadlines are missed. Therefore, a high $MPCG$ is beneficial for the GPU utilization (since the command queue rarely runs empty) but might result in more deadline misses of applications with high priority.

4.4. Implementation

In this section we describe the implementation of our scheduling framework and the interfaces of the scheduler module to the native driver and the compositor. The patch[3] to the native Vivante kernel driver consists of 3036 lines, more precisely, it adds 2167 lines of C code, changes 78, and removes 5 (determined with "cloc --diff"). Our GPU driver module consists of 3155 lines of C code and 735 lines of source code comments (determined with "cloc"). This includes a short implementation (32 lines) of the FIFO algorithm, which can be optionally activated and was used only to evaluate the execution time prediction presented in Chapter 3.

4.4.1. Hardware platform and Operating System

In the automotive cockpit demonstrator VCT-B presented in Sec. 2.3, a Freescale i.MX6 platform is used. To this end, we implemented the GPU scheduling concept described in Sec. 4.3 for the same platform. It contains four ARM CPU cores, a Vivante GC2000 GPU for 3D rendering, a Vivante GC355 GPU for 2D compositing, and an IPU. As operating system we used Yocto 1.8 with Linux kernel 3.14.28-rt25 with preempt-rt patches.

4.4.2. Dispatching commands

The native user space driver uses system calls to submit different types of commands to the kernel space driver. The majority of these commands are not executed by the GPU, and thus do not have to be dispatched by the GPU scheduler; for instance, commands query driver information, allocate memory, or alter driver behavior. The commands that need to be dispatched by the scheduler are the submission of:

- Synchronization points (called *Sync Events*)
- GPU Command Groups (*CGs*)
- Detach commands if a process explicitly disconnects

Without our modifications, the native GPU driver synchronously dispatches those commands in system call context. This means, that after having

[3] Created with "diff -Naurp --exclude=*.out --exclude=.cproject --exclude=.project --exclude=.settings --exclude=Makefile --exclude=*.txt --exclude=*.swp orig_imx_3.14.28_1.0.0_ga gpusched_imx_3.14.28_1.0.0_ga"

submitted a CG, the application is blocked until the CG has been dispatched. Simple scheduling approaches like [KLRI11, BDC08, YZQ+13] delay this dispatching individually for the respective processes. However, such a synchronous scheduling is inherently unaware of more than only one undispatched CG, whereas each frame is rendered by multiple CGs. This is a major drawback, which justifies our approach of asynchronous dispatching. More precisely, the scheduler does not know the predicted execution time of a complete frame until its last CG has been submitted (i.e., the CG containing the SwapBuffers GPU command). Therefore, the execution time needed to render the whole frame is estimated by means of the demanded execution time per frame (*etpf*) of an application (cf., Sec. 4.3.2). However, the *etpf* can be smaller than what the application submits (e.g., *etpf*=0), which would inevitably result in not dispatching lower-priority applications or potentially cause deadline misses of higher-priority applications. Additionally, the *etpf* can be bigger than what an application submits, in which case lower-priority applications could not benefit if the scheduler does not know it. Therefore, we implemented asynchronous dispatching where—in the system call context—the CG is just forwarded to the GPU scheduler, which enqueues it into the application's scheduling queue, and then the system call immediately returns. Thus, the application is able to submit its CGs as fast as possible without having to wait until they actually get dispatched. The GPU scheduling algorithm, which runs in a dedicated kernel thread, later dispatches the CG from the scheduling queue. A major difficulty of the implementation was, to allow dispatching by a process different than the originating process and at an unexpected point in time. In more detail, the dispatch function expects that it has access to a couple of data structures allocated by the user space application. When the system call returns, the user space application might free this data or reuse it for other purposes. While the process is in system call context, we therefore create copies of the relevant memory blocks and use these copies later while dispatching. This includes the CG's command buffers, the Sync Event queues, and the GPU State Deltas[4]. Thus, instead of performing the memory copy operations in the dispatch function, we moved the copy operations ahead. When dispatched by the scheduler thread, the dispatch function then just uses the existing copies. Where possible (i.e., where the data size is constant), we

[4] A State Delta is a set of GPU commands updating GPU-internal state registers and is used to update the Context Buffer associated with the application. The Context Buffer is executed each time a GPU context switch occurs.

used the Linux kernel Slab allocator. Additionally, we make dispatching aware of the effective process and thread id, which is obviously no longer the current real process or thread id, since the scheduler kernel thread is now the caller. Note, that asynchronous dispatching is about as fast as concurrent synchronous dispatching, since most of the dispatch code is protected by mutexes, which prevent concurrent execution anyway.

4.4.3. Time measurement and prediction

In order to do real-time scheduling, we need to know the exact time when each of the CGs started and finished execution. The GPU operates asynchronously to the main CPU. To synchronize CPU execution with GPU execution, typically interrupts are used. To this end, dedicated GPU commands are used, which—when executed—emit an interrupt. These commands are called *Sync Events* and used whenever the application (e.g., for glFinish()) or the driver internally (e.g., to enter power saving state) need it. We patched the GPU Driver (kernel space), so that after each execution of a CG, a Sync Event is submitted. However, dispatching one CG can submit multiple Sync Events, which necessitates counting the number of submitted Sync Events and to precisely count the number of received interrupts to determine when each CG is finished. Moreover, dispatching (and thus Sync Event submission) takes place concurrently to GPU execution, such that when receiving an interrupt, we do not know whether the CG has finished, since the corresponding dispatch could concurrently submit more Sync Events. Therefore, we notify the scheduler module right before the last Sync Event of the CG is submitted to the GPU. The scheduler then marks the number of expected interrupts as complete, and—after receiving the expected number of interrupts—the CG is marked as finished, with the correct finish timestamp, obtained using the *cpu_clock(0)* function.

4.4.4. GPU Scheduler interface

Our patch to the 3D GPU Driver kernel module "galcore" allows a separate GPU scheduler kernel module to be loaded and hooked into its functionalities. Next, we describe which functions of the driver are accessible by the GPU scheduler, then we describe how the scheduler module attaches to the driver and which callbacks it uses.

The GPU scheduler uses multiple new, exported functions of the driver to interact with the GPU. When the module is loaded, initially *mxc_gpusched_register* is called, which enables the GPU scheduling mode of the patched native driver. To dispatch a CG, the scheduler calls *mxc_gpusched_dispatch*. After the user space process has exited and all of its previous CGs were executed, the GPU driver calls *mxc_gpusched_detach_process*, which frees all data related to that process. Eventually, when the scheduler module is unloaded, *mxc_gpusched_unregister* is called to notify the driver that the GPU scheduler mode shall be disabled. As depicted in Fig. 4.2, after the GPU scheduler module is loaded, the driver forwards all new CGs received by applications directly to the scheduler instead of immediately dispatching them. The GPU scheduler is then called by the driver at multiple occasions, which are, grouped by origin, depicted in Fig. 4.9. For each type of occasion, the scheduler module implements a dedicated callback method. An application that wants to access the GPU for the first

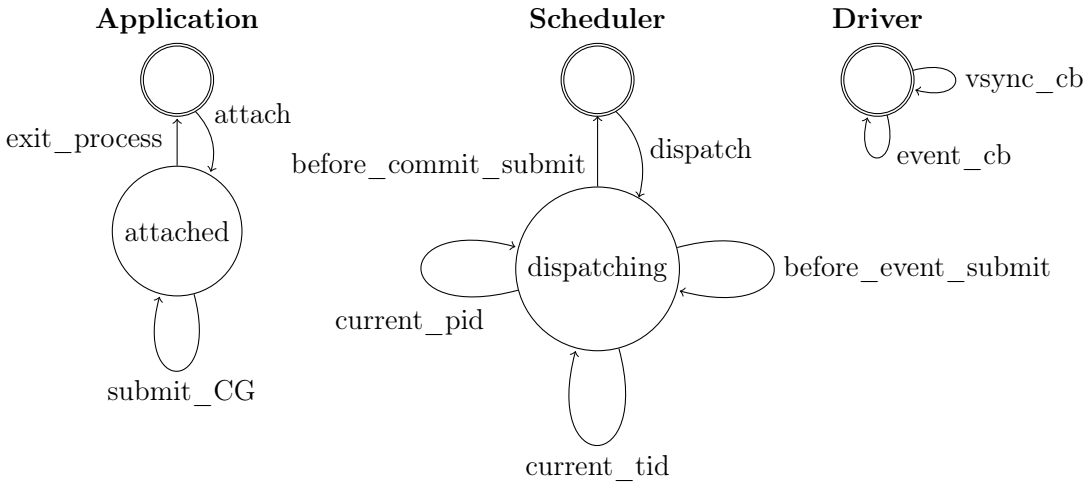


Figure 4.9.: Scheduler interface callbacks

time, makes a system call that attaches it to the kernel space driver. Then, this application is known to the driver, and, also, due to the callback, to the GPU scheduler. The application can now submit CGs, which are forwarded to the GPU scheduler via the callback *submit_CG*. When the process exits or gets killed, the GPU scheduler—after all its commands have finished—releases its resources. The scheduler calls the dispatch function of the driver to submit a CG to the GPU. While the dispatch function executes, the driver gives a callback for each Sync Event submitted to the GPU. This is required, since this directly corresponds to the number of interrupts that are emitted by the GPU

and the scheduler has to count that many interrupts before it can detect that this CG is finished. Furthermore, the native driver at some places queries the pid (process id) or the tid (thread id) on which behalf it schedules. Directly before starting the main execution of the CG, another callback tells the GPU scheduler when exactly the CG was submitted. At any point in time, the driver can notify about interrupts emitted by the 3D GPU (*event_cb*) or by the display device driver (*vsync_cb*).

4.4.5. Compositor interface

For compositing, we implemented a simple interface, which consists of a system call that waits until the next vsync event occurs. The Compositor passes a pointer along with the system call. Before the call returns, our implementation copies the list of all windows that were completed before the vsync event to this pointer address. The Compositor then takes this list and bitblits each window at the desired place. Although compositing is not the main focus of this work, we implemented a full compositing approach to demonstrate that compositing easily integrates with our GPU scheduler. The compositor can be built to run in stand-alone mode on a native Linux or integrated into our automotive cockpit demonstrator VCT-B described in Sec. 2.3.

4.4.6. Concurrency

In our implementation, many threads run concurrently and therefore need to be synchronized. Implementing this in an efficient way was a major challenge. In Fig. 4.10 we have depicted the different components and the events they are signaling or waiting for. For all of them we used the *completions* from the Linux kernel, which provide better efficiency than semaphores. An application thread

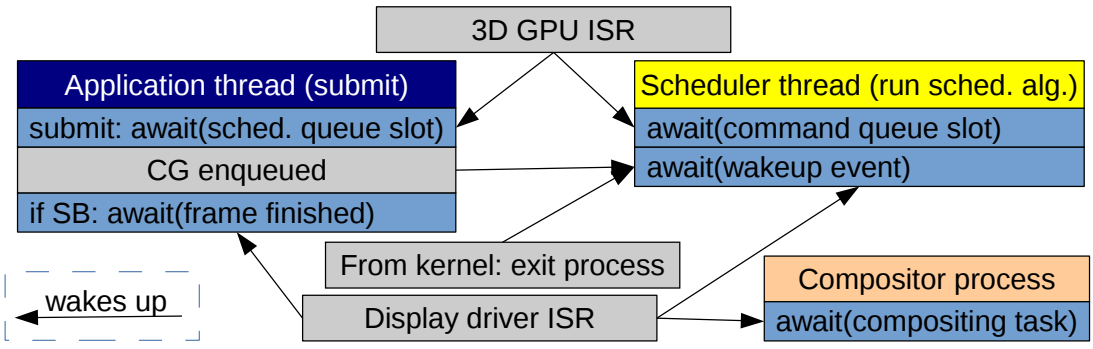


Figure 4.10.: Scheduler thread concurrency synchronization

that submits a new CG is blocked, if its associated scheduling queue, which holds 64 entries, is full. After the GPU has finished a CG, it potentially wakes up the waiting application thread. The size of the command queue *MPCG* limits the number of pending CGs. The scheduler thread runs in a loop where it first waits until the command queue has free slots, then it waits for a wake-up event and eventually executes the scheduling algorithm described in Sec. 4.3.3 (Listing 4.3). The scheduler waits for a wake-up event only if in its previous run no schedulable CG was found, which means that further loops of the algorithm would also fail. To this end, the scheduler thread sleeps until the next frame period starts or a new CG is enqueued, either directly by the application or by the kernel on process exit. After the application has inserted a SwapBuffers CG, it sleeps until its next incipient arrival time (cf., Listing 4.2, Line 4). Additionally, the compositor process waits for a vsync event where application windows are waiting to be bitblitted.

Besides the *completions* mentioned before, we occasionally use atomic variables and memory barriers, where necessary. Additionally, we replaced all *mutex* instances by *rt_mutex* instances, which provide priority-inheritance, and significantly reduce latency jitter. This was primarily relevant for debugging output in both, the driver and the scheduler module, since all debug output is serialized using a global mutex. Two special concurrency issues are explained next. The driver uses a global mutex that protects each access to one of the driver's context data structures. Contexts are accessed when a process attaches or detaches and also while dispatching commands. With GPU scheduling this is a drawback, since an attaching process can temporarily delay the scheduler kernel thread that dispatches or executes a detach command. In our algorithm, this would increase the delay of scheduling and dispatching by a relatively high and hard to estimate amount of time. We therefore decided to disable this mutex completely while in scheduler mode. This is possible, since attaching creates a new context, which cannot cause a conflicting access. All other places are executed only by the scheduler thread, which means that no concurrency can occur while in scheduling mode.

A similar drawback exists with the global event queue mutex. It is locked when new Sync Event objects are reserved, before the Sync Event command is sent to the GPU, and also after the GPU has executed the Sync Event and the associated actions are executed. Executing the associated actions is performed by a worker thread of the native driver that is triggered by the ISR. If GPU scheduling is active, Sync Event reservation is done by the scheduler kernel thread. Thus,

the worker thread could delay the scheduler kernel thread by hardly predictable amounts of time. We replaced this mutex by a lock-free implementation that uses a modified execution sequence and one read and one write memory barrier.

For the sake of clarity, we omitted a few implementation details, which are not essential to understand how the scheduling algorithm works. The omitted details include:

- Handling of boundary cases (e.g., introduced by integer rounding)
- Assertions that verify the correctness of the scheduler execution
- Tracing of scheduler decisions in a dedicated data structure
- A dedicated scheduling queue for each priority, the attached ring buffer of CG entries, and the dedicated data structure for the GPU execution flow are all separate data structures, but shown as Q in the pseudo code
- Control of the scheduler process and application processes (wait and wake up).
- The special handling of "exit" CGs that free the application's GPU driver resources of the native GPU driver. It does not involve the GPU but nevertheless cannot be dispatched out of order.

4.5. Evaluation

In this section, we evaluate the effectiveness, the achieved GPU utilization, and the efficiency of our scheduling approach. Effectiveness means that the scheduler enforces the applications' priorities and desired *frame rate*s. We evaluated simple homogeneous scenarios, as well as sophisticated mixed scenarios. The achieved GPU utilization is also evaluated for the mixed scenario, using different values for the *MPCG*, since the command queue length affects the utilization. The efficiency evaluations determine the CPU overhead introduced by our scheduling algorithm. In this case, simple homogeneous scenarios are used, since the actual rendering of the applications does not affect the CPU overhead of the scheduling algorithm.

4.5.1. Setup

As hardware platform for our evaluations we used a Freescale (NXP) i.MX6 SABRE Automotive Platform. The board features a quad core ARM CPU running at 800 MHz and 2 GB of RAM. Its SOC contains a Vivante GC2000 GPU for 3D rendering, and a Vivante GC355 GPU for 2D compositing. The Freescale (NXP) i.MX6 platform is widely used for automotive multimedia and infotainment applications (e.g., [Con15, AUD14]) and thus is a representative automotive platform for realistic evaluations of our concepts. We used a Yocto 1.8 system image based on Linux kernel 3.14.28 with preempt-rt patches and the Vivante driver V5.0.11.p4.25762. The execution time *predET*—predicted by libETP—was used to calculate (with numbers based on evaluations)

$$predET_C = (predET + 100\,\mu s) * 1.05 \qquad \text{and}$$
$$predET_O = MAX(0, predET - 100\,\mu s) * 0.9$$

(cf., Sec. 4.3.5). As smallest possible execution time of a CG we used $min_{et} = 30\,\mu s$, by using pc_{flush} (cf., Sec. 3.8.1.3) with a small safety-margin. For the conservative and optimistic estimation of the delay of scheduling and dispatching we used $\texttt{SDdelay}_C = 150\,\mu s$ and $\texttt{SDdelay}_O = 50\,\mu s$, based on our evaluation results (Fig. 4.18). By default, we used $\texttt{MPCG} = 5$ (i.e., up to five pending GPU commands were allowed), $\texttt{FPLA} = 7$, and evaluated each scenario for at least 300 s. In the presented results, we skipped the first 50 s and the last 10 s, since the applications rarely submitted CGs while initializing and while terminating. The priorities of the daemon threads of the native GPU kernel driver were increased

from 0 to 40. The ISR of the 3D driver was executed with priority 95, while our scheduler kernel thread had priority 46. All user space applications were running at default priority and "nice" level. The 3D applications were using OpenGL ES 2.0 and EGL with the Vivante framebuffer backend as graphics APIs and used libETP. The application-specific scheduling queues hold up to 64 CGs. Since power-management of CPU and GPUs potentially increases latency, it was disabled.

4.5.2. Effectiveness

In this subsection, we evaluate the effectiveness of our scheduling approach in order to verify whether given deadlines are met for high-priority applications, while low-priority applications can utilize the remaining GPU resources. We used a set of identical applications, namely the glmark2-es2 "build" benchmark scene rendering the "bunny" model, which has a fast CPU execution time and a precisely predicted GPU execution time. This ensures that applications do not miss deadlines just because they submitted their CGs too late due to non-prioritized CPU scheduling. We executed 10 applications in parallel, each with *etpf*=5 ms. In Fig. 4.11, we depict the results for *frame rate*=60 FPS. For each priority (corresponding to a process of the application executable), we show the percentage of met deadlines and the achieved *frame rate*. We observe

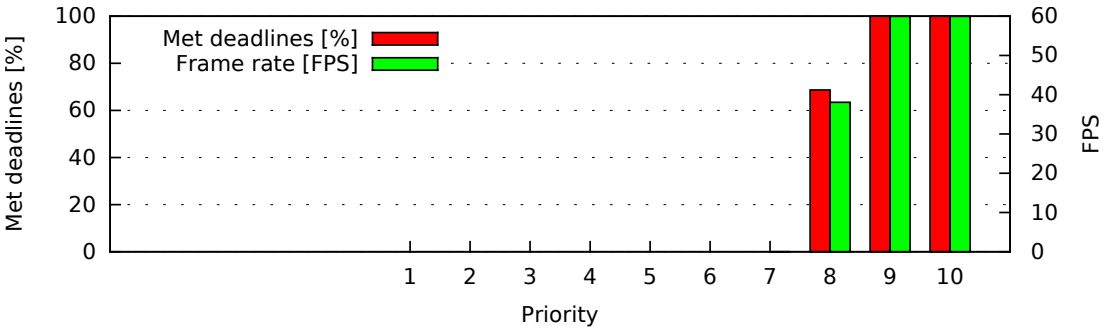


Figure 4.11.: Effectiveness (homogeneous scenario), 60 FPS

that the deadlines of the priorities 10 and 9 were always met and the desired *frame rate* of 60 FPS was fulfilled. Priority 8 met its deadlines in about 70 % of the cases where its CGs interleaved into the CGs of the Priorities 10 and 9. Priorities lower than 8 were never admitted due to the high amount of execution time reserved for higher-priority applications. We additionally evaluated the same scenario using 30 FPS for each application, cf., Fig. 4.12. As expected, the *frame rate*s dropped to 30 FPS. Now, 6 applications (Priorities 10 through 5)

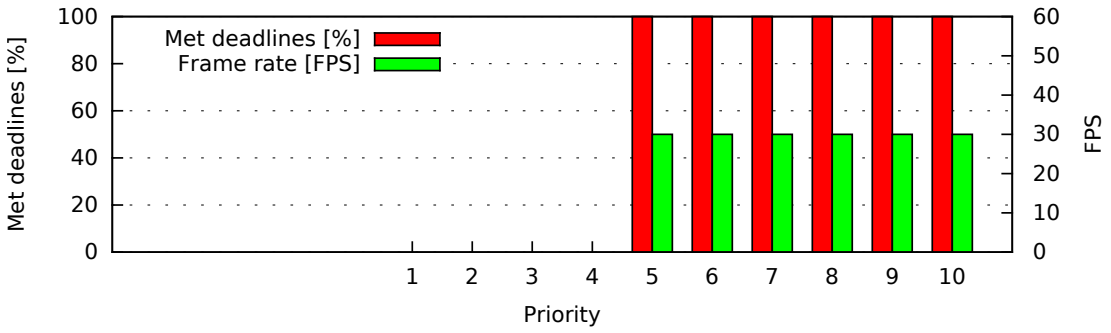


Figure 4.12.: Effectiveness (homogeneous scenario), 30 FPS

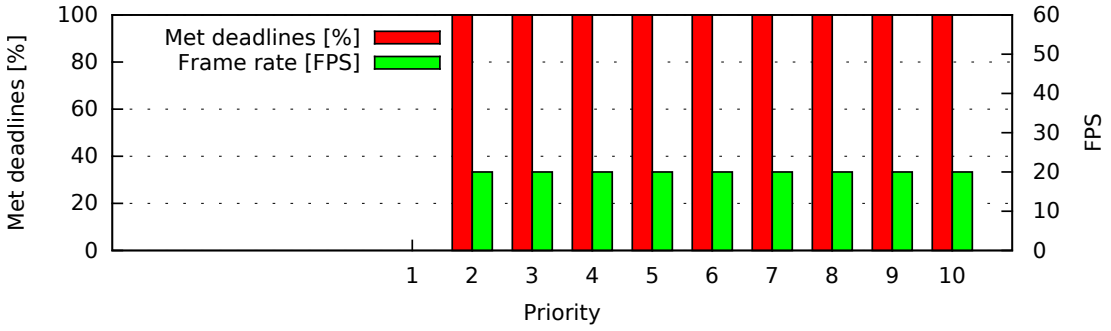


Figure 4.13.: Effectiveness (homogeneous scenario), 20 FPS

met all deadlines, since the GPU scheduler now interleaves the CGs in intervals of 33.3 ms (2 periods) instead of 16.7 ms (1 period). Thus, the number of applications with fully met deadlines more than doubles compared to 60 FPS. The scheduler effectively improves interleaving of the CGs and admits more CGs. Additionally, if all applications have 60 FPS, during a short period of time directly after a new vsync period starts, all scheduling queues are empty, thus making the GPU idle for about 400 μs on average.

Moreover, we evaluated the same scenario using 20 FPS for each application, cf., Fig. 4.13. As expected, the *frame rate*s dropped to 20 FPS and 9 applications were dispatched and met all their deadlines. Changing from 30 FPS to 20 FPS increases the number of dispatched applications from 6 to 9. The evaluation results of the homogeneous scenario demonstrate that the scheduler enforces the scheduling according to the defined priorities, *frame rate*s, and *etpf*.

We also evaluated a more sophisticated, mixed scenario, consisting of the set of applications and parameters depicted in Table 4.1. The mixed scenario uses 17 concurrent applications. This includes Quake 3, which submits many CGs per frame and occasionally faces significant prediction errors (cf., Sec. 3.8). The selected set of applications ensures that the scenario is always GPU-bound by using 10 instances of the Glmark2-es2 "texture" program with low priority. In

Table 4.1.: Application setup for mixed scenario

Priority	*frame rate*	*etpf*	Application type	Resolution
17	60 FPS	2.1 ms	Automotive speedometer	456 × 456
16	60 FPS	2.1 ms	Automotive tachometer	456 × 456
15	30 FPS	2.3 ms	Glmark2-es2 “shading”	720 × 540
14	20 FPS	2.5 ms	Glmark2-es2 “texture”	720 × 540
12 & 13	30 FPS	0.0 ms	Glmark2-es2 “build” (“bunny”)	720 × 540
11	30 FPS	0.0 ms	Quake 3 “demo four”	1440 × 540
1 – 10	60 FPS	0.0 ms	Glmark2-es2 “texture”	720 × 540

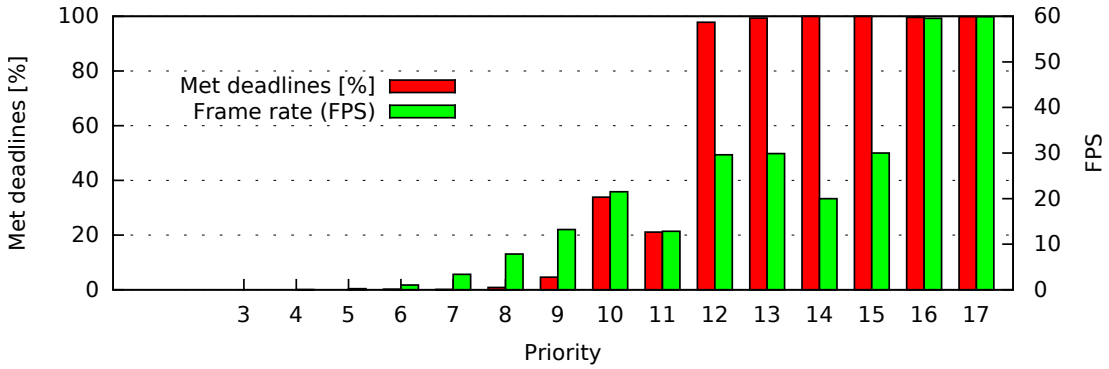


Figure 4.14.: Effectiveness (mixed scenario)

Fig. 4.14, we depict the results for the effectiveness at the mixed scenario. We observe that the priorities 17 to 12 met most of their deadlines. The few missed deadlines (especially of Priority 12) were caused by the Execution Time Predictor, which occasionally underestimated the CGs of Quake 3, by a couple of milliseconds. Since the scheduler typically schedules higher priorities first, in the rare situation that not all deadlines of the Priorities 12 through 17 would be met, at least Priority 12 would miss its deadline. While this is not the fault of the scheduling algorithm, it makes clear, that accurate execution time prediction is important. Next, we discuss the results in more detail.

The Priorities 17 through 14 met more than 99.6 % of their deadlines. Priority 17 met 99.89 % of its deadlines, while Priority 16 met 99.62 %. The Priorities 14 and 15 even met 100 % of their deadlines, since their deadlines were later, due to a lower *frame rate*. To a minor degree, the Priorities 13 and 12 suffered also from *etpf* = 0, since for future frames no execution time was reserved for them. The priorities below 11 still got GPU resources, namely the resources that could not be used by Quake 3 (Priority 11) due to the following two reasons. First, when long running or overestimated CGs of Quake 3 could

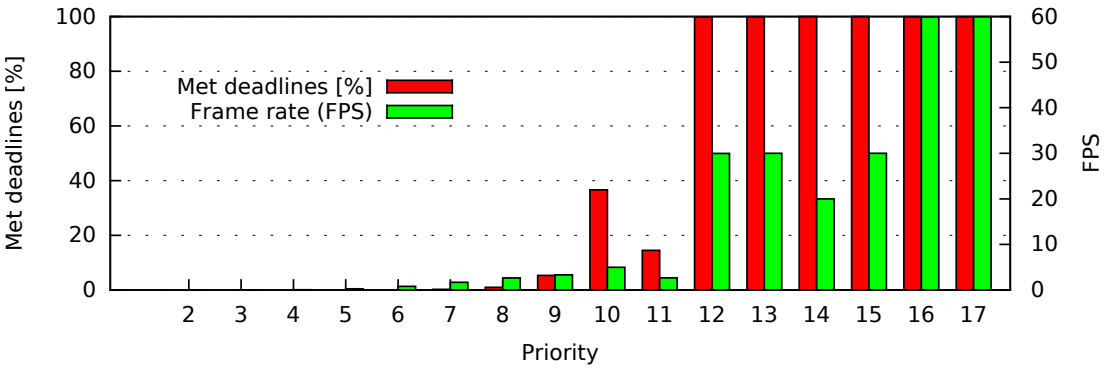


Figure 4.15.: Effectiveness (mixed scenario), Quake 3 with 200 % *predET*

not be scheduled, since deadlines of the Priorities 17 through 12 could be violated, a lower priority CG that could not cause a deadline miss of Quake 3 was scheduled, as explained in Sec. 4.3.5.3. Second, Quake 3 is often CPU bound, especially during the loading of a scene. If the scheduling queue of Quake 3 is temporarily empty, due to *etpf*=0, the scheduler skips the scheduling queue of Quake 3 and—if no deadline of the Priorities 17 through 12 is violated—dispatches a CG with lower priority. The embedded GPU and CPU are too slow for Quake 3, since even if running stand-alone, Quake 3 achieves only about 20 FPS[5]. In the mixed scenario, Quake 3 therefore gets a fair share of the GPU resources and can render at 12.8 FPS.

While the amount of met deadlines is affected—besides the priority—by the *frame rate*s and the *etpf*, the predominant impact on the effectiveness of the scheduler is the prediction error of Quake 3. Hence, we additionally evaluated the same mixed scenario, where libETP predicted always 100 % more[6] for all CGs of Quake 3. In Fig. 4.15, the results are depicted. Since often the CGs of Quake 3 would violate higher priorities, the framerate of Quake 3 drops to 2.7 FPS. The positive effect is, however, that the higher-priority deadlines of the Priorities 17 through 12 are very rarely missed, now (at least 99.85 % of the deadlines were met). The highest Priorities 17 went up to more than 99.97 % of met deadlines. Since Quake 3 was estimated higher, lower priorities were still dispatched, but less often, due to the reservations made for Quake 3. Moreover, predicting higher execution times by libETP for selected unknown applications is a good way to ensure that high-priority deadlines are met even if execution time prediction has limited accuracy. Appendix A.3.2 contains additionally the results for an estimation of ∞ and an underestimation by 75 % for Quake 3.

[5]During normal scene rendering, not during the loading of a scene
[6]This was achieved using the OVERPREDICT_FACTOR parameter of libETP, cf. Sec 3.7.5

4.5.3. GPU Utilization

In this section, we evaluate the achieved GPU utilization. We denote the GPU utilization $GPU_{util}(t_{from}, t_{to})$ as the time the GPU is effectively busy during period $[t_{from}, t_{to}]$ in relation to the length of this period. To measure GPU_{util}, we used again the set of applications depicted in Table 4.1 and varied the maximum number of pending CGs (*MPCG*). The results are depicted in Fig. 4.16. In this figure, each point represents GPU_{util} for a duration of 2 s. The

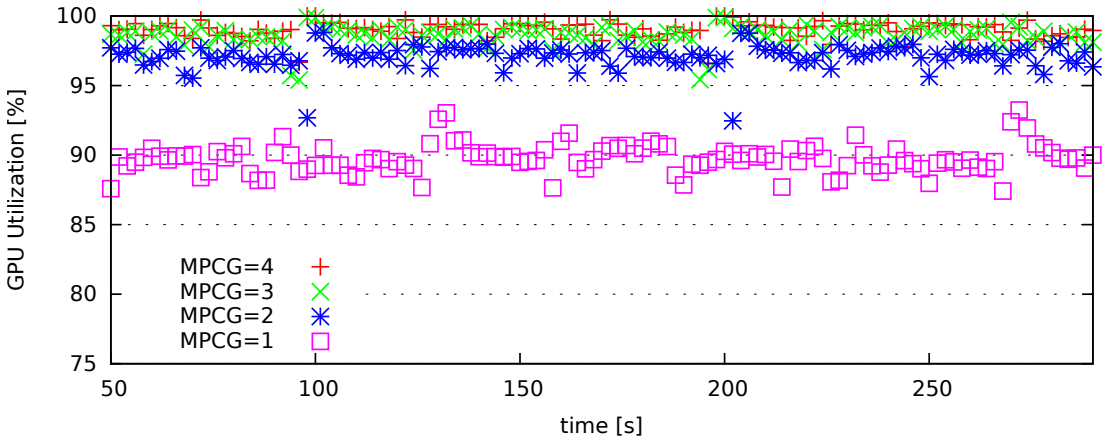


Figure 4.16.: GPU utilization, mixed scenario

possible interleaving influences the utilization and results in a mediocre fluctuation. However, at two occasions, the points of *MPCG* 2 through 4 slightly drop, since during those periods Quake 3 is loading a new scene and submits CGs with very small execution time. Whenever the command queue runs empty, the GPU goes idle, which happens more often if many fast CGs are dispatched. Therefore, using a different set of applications could prevent these drops in GPU utilization.

In Fig. 4.17, the average GPU utilization is depicted. The GPU utilization with `MPCG` = 2 is significantly higher (97.11 %) than with `MPCG` = 1 (89.75 %), since the scheduler can run in parallel to the GPU execution. In situations where the dispatched CGs are very fast (in the range of tens of micro seconds), `MPCG` = 3 additionally increases GPU utilization to 98.6 %. Values for `MPCG` higher than 5 provide almost no gain in GPU utilization, since the scheduling algorithm is very fast. On average, a GPU utilization of at least 99 % was achieved for `MPCG` = 4 through `MPCG` = 10. In Fig. 4.17, additionally the average number of scheduler runs per dispatched CG are depicted. If *MPCG* is increased, the *busyUntil* values of the scheduling algorithm also increase, since, typically, more CGs are pending. This increases the probability that at *tRef* (which is based on *busyUntil*), no CG

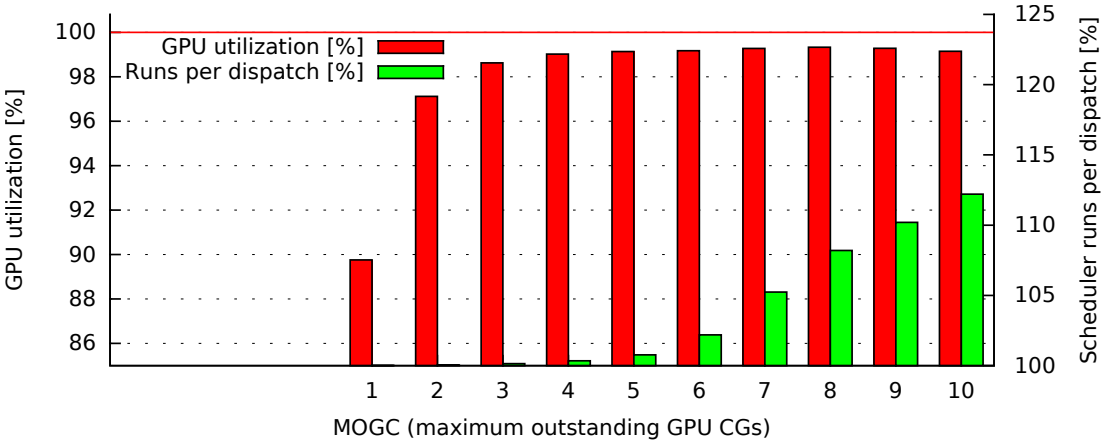


Figure 4.17.: Average GPU utilization and required number of scheduler runs

can be dispatched without violating a higher-priority deadline, i.e., *schedule_next* terminates with $Qdis = 0$ (cf., Listing 4.3). The CPU overhead introduced by a number of scheduler runs exceeding 100 % is negligible for `MPCG` <= 5. For `MPCG` = 5, the overhead is only 0.77 %, but rapidly grows for higher values. A `MPCG` below 5 has a noticeable lower GPU utilization and a `MPCG` above 5 increases the CPU overhead without clearly increasing GPU utilization. Plainly, there exists a trade-off between high GPU utilization and low CPU overhead of the scheduler. Since higher values of *MPCG* accumulate errors of the execution time prediction (cf. Appendix, Sec. A.3.1) and we opt for a compromise between high GPU utilization and low CPU overhead, `MPCG` = 5 was used as default value in our evaluations.

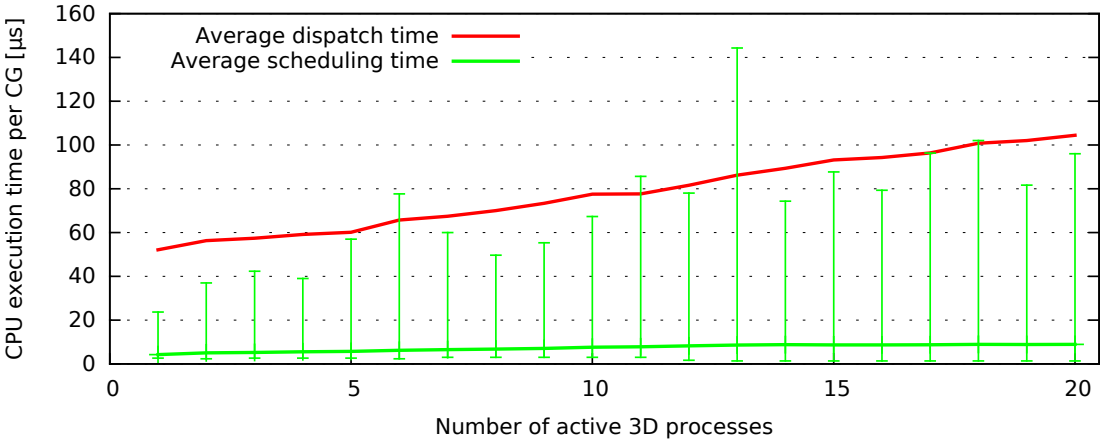


Figure 4.18.: Delay of the scheduling algorithm

4.5.4. Scheduler Efficiency

Finally, we evaluate the CPU execution time consumed by the scheduling algorithm. We used a varying number of "es2gears" applications at a resolution of 320x240, 30 FPS, and $etpf = 1.0\,\text{ms}$[7]. In Fig. 4.18, we show the average execution time of the scheduling algorithm and minimum and maximum values. We observe that the execution time shows a linear dependency on the number of active processes with a small slope. On average, the scheduling algorithm stayed always below 9 µs. It never exceeded 145 µs (single outlier). In contrast, the execution time needed to dispatch a CG increased on average from 52 µs for a single application to about 104 µs for 20 applications. This is caused by data structures of the native GPU driver where the CPU execution time for lookups depends on the number of data records, which grows with the number of rendering threads[8]. Thus, our scheduling algorithm adds typically less than 10 % to the CPU time needed for dispatching by the native driver, which is a quite impressive result. More precisely, in our mixed scenario, our scheduling algorithm consumed 1.0 % of the CPU time of a single CPU core, while the native GPU driver consumed 12.8 %. Since the evaluation platform has four CPU cores, our GPU scheduler required only 0.25 % of the CPU resources.

4.5.5. Evaluation conclusion and summary

The goals for automotive GPU scheduling are (a) fulfill deadlines based on the *priority*, (b) guarantee the desired *frame rate*, (c) achieve a high GPU utilization, and (d) achieve a low CPU overhead (cf., Sec. 4.1).

Our evaluations address all four goals: (a) and (b) are addressed by the effectiveness evaluations (Sec. 4.5.2), (c) by the utilization evaluation (Sec. 4.5.3), and (d) by the efficiency evaluation (Sec. 4.5.4).

Our evaluations for effectiveness show, that our scheduler correctly handles priorities and desired *frame rate*. In the three evaluations for the homogeneous scenario, the high priorities met their deadlines by at least 99.98 %. This applies to the two highest priorities with a *frame rate* of 60 FPS, the six highest priorities with a *frame rate* of 30 FPS, and the nine highest priorities with a

[7] es2gears was selected because it allocates only a small amount of memory, thus allowing to run many instances on our memory-limited embedded platform.

[8] This fact can also be observed the Appendix, Fig. A.5, where the delay introduced by the scheduling algorithm (depicted in green at SCHED, almost invisible) is almost negligible compared to the delay introduced by the dispatching performed by the native Vivante driver (depicted in blue at SCHED).

frame rate of 20 FPS. In our mixed scenario, we show that the scheduling algorithm also works with different *frame rate*s, with and without using the *etpf*, and the impact of an application that has a high execution time prediction error (namely, Quake 3). Even with the high prediction errors of Quake 3, the biggest impact (Priority 12 in Fig. 4.14) was still a very high rate of met deadlines of 97.8 %. We showed, that Quake 3 is the only reason for these deadline misses. In an automotive scenario, the critical applications can be developed in a way that the execution time can be predicted accurately, e.g., by using different shaders for different scenes and choosing the best models. In contrast, applications of low criticality (such as Quake 3) that might show higher prediction errors, have still a quite limited impact on the scheduling of the critical applications. If required, a safety margin can be added to the predicted execution time for applications with potentially higher prediction errors, which was demonstrated for Quake 3 (results in Fig. 4.15).

Our concept of deadline- and priority-based scheduling achieved a very high GPU utilization of more than 99 % on average. Our results show the command queue affects GPU utilization as well as CPU overhead of the scheduling algorithm. For our mixed scenario, a value of `MPCG` = 5 was the best trade-off, since GPU utilization does not increase with a higher `MPCG`, while the scheduling overhead increases, since with a long command queue the scheduling algorithm often runs without being able to dispatch any CG without violating future high priority deadlines.

Moreover, we showed by evaluations, that the CPU overhead introduced by running the scheduler is minimal. In our complex mixed-scenario, where the scheduler did run about 1500 times per second on average, it only consumed 0.25 % of the total CPU resources, while the native GPU driver consumed 3.2 %. Even with 20 concurrent application, a run of the scheduling algorithm did not take longer than 9 µs on average. Thus, our scheduling algorithm is not only effective, but achieves a very high GPU utilization with little CPU overhead.

4.6. Outlook on preemptive scheduling

As motivated in Sec. 1.2, current GPUs do not provide sufficient preemption-capabilities to support preemptive real-time scheduling. Nevertheless, in this section we briefly discuss the relevance of our concepts and results for a preemptive 3D GPU scheduling, which might become possible in the future.

In automotive scenarios, applications have different priorities, different frame strides, and can submit a non-fixed sequence of GPU CGs per frame. Thus, if GPU preemption should be available, a preemptive GPU scheduler would still have to support such scenarios. The well-known EDF (earliest deadline first) scheduling is insufficient, since it does not consider priorities. FPPS (fixed priority preemptive scheduling) can be used and would schedule the highest-priority application wherever possible. However, all other applications might be scheduled never or rarely, which implies that the GPU utilization might be very low. To explain in more detail why FPPS would suffer from low

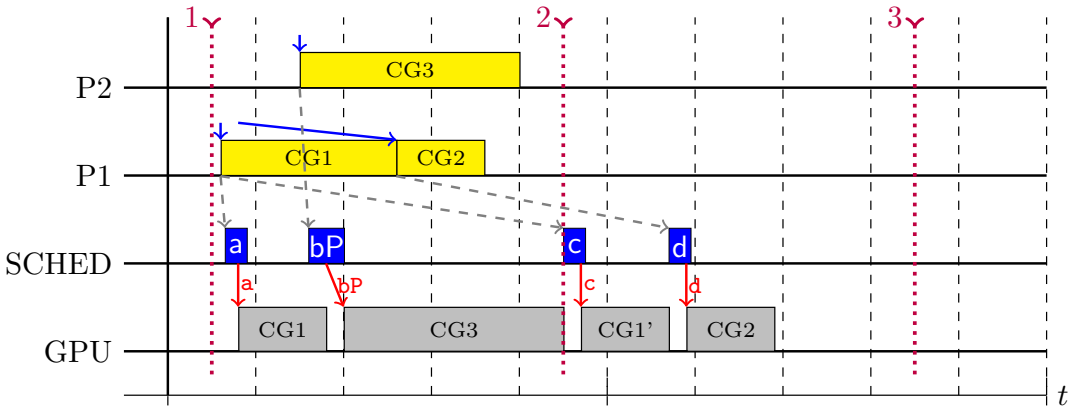


Figure 4.19.: Example for simple priority-based scheduling

GPU utilization, a short example with two applications using FPPS is depicted in Fig. 4.19. The higher priority P2 has its next deadline at frame sequence number 3 (red dotted line), while the lower priority P1 has its next deadline at frame sequence number 2. With FPPS, P1 is scheduled first, since its CGs arrived first. As soon as CG3 from P2 is received, it is dispatched, and, consequently, CG1 is preempted. After CG3 has finished, the scheduler dispatches the remainder of CG1, followed by CG2. In this example, P1 missed its deadline. While FPPS always ensures the highest priority does not miss its deadline, it does not share the goal of dispatching as many lower priority

applications as possible. In order to overcome this significant drawback, the scheduling algorithm needs the predicted execution time and (ideally) the *etpf*. Moreover, preempting the GPU takes some time. It is therefore beneficial to keep the number of preemptions small; since our scheduling algorithm is optimized for GPUs that do not support preemption, preemption would be required very rarely, thus ensuring a high GPU utilization. Additionally, other concepts presented in this work, such as the asynchronous scheduling queue, also improve preemptive scheduling.

To this end, GPU preemption could be used to extend our scheduling algorithm such that the available time is always used for the lowest possible priority that does not violate higher-priority deadlines. This can increase the GPU utilization and the effectiveness, since some deadline misses of lower-priority applications—especially those with long-running CGs—can be avoided.

4.7. Related Work

Real-time GPU scheduling as targeted by our approach and classic real-time scheduling for CPUs [LL73, RBE99, JRR97] 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), supports GPU preemption since Version 1.2 [WDDa], which was introduced with Microsoft Windows 8, preemption is de facto an optional feature[9] and an upper bound on preemption latency is not guaranteed. Actually, many GPUs—such as the Vivante GC2000 used in this work—do not support preemption[10], so a command batch cannot be preempted once it was started. A few recent GPUs support preemption only on DRAW level [RS15]. This means that if the operating system issues a preemption request, the GPU must finish the execution of the current DRAW and can then be switched to a more important 3D context. Depending on the API usage pattern of the 3D applications, this can significantly reduce the latency between preemption request and switch to another context. However, the execution time of a single DRAW call is not limited, since the number of vertices and fragments, and the execution times of the shader instances can be (in theory) arbitrarily high. The Nvidia Kepler GPUs (available since Q3/2016) introduced preemption on fragment-level[11] claiming a latency of less than $100\mu s$ [Bou16]. However, the execution time of a single fragment shader is (in theory) not limited, which clarifies that the given latency is not guaranteed at all. Additionally, Nvidia does not state preemption of other steps of the rendering pipeline, e.g., if fragment shaders or rasterization can also be preempted. When performing fragment-level preemption (which probably includes vertex-level preemption), the whole state of the hardware rendering pipeline (except for the shader core control blocks), which uses many caches [Smi16], must be saved to be able to continue later. Such a save and load feature increases the die size of the GPU, which is probably the reason why no embedded GPUs support fragment-level preemption, yet. The only effective way to preempt the current work of a GPU

[9] [WDDa] states that "WDDM 1.2 drivers can also reject the Windows 8 preemption model"

[10] However, DRAW-level preemption (explained thereafter) could probably be implemented in the kernel driver, given that the command set of the GPU is known (e.g., through reverse engineering).

[11] Actually, they announced "First ever pixel-level graphics preemption", i.e., fragment-level preemption.

with limited latency is to reset it, e.g., by power-cycling it. After resetting, the GPU and the kernel driver must be reinitialized, which typically means that GPU contexts are lost and the respective applications must reinitialize and render again their content [WDDb]. To this end, resetting takes typically many seconds and is unsuitable for automotive use cases with critical applications. Nevertheless, even if preemption should become available in the future, our concepts provide a major contribution (cf., Sec. 4.6).

Real-time GPU scheduling must guarantee the timely execution of rendering tasks. In general, preemption is one of the basic mechanisms that facilitate the implementation of scheduling concepts. Although preemption mechanisms are foreseen by GPU driver frameworks such as the Windows Display Driver Model [WDDa], no maximum delay between preemption request and completion is guaranteed, making them insufficient for the implementation of real-time GPU scheduling. On the other hand, non-preemptive scheduling approaches typically assume a static set of applications and time requirements. Bautin et al. [BDC08, DWA08] developed a system called Graphics Engine Resource Manager (GERM), which targets fairness in GPU multitasking, using deficit-round-robin scheduling. However, they do not directly support priorities but assume that increasing an applications operating system priority indirectly allows it to submit CGs at higher frequency. Furthermore, this approach does not support frame-based deadlines nor GPU execution time reservation.

Kato et al. developed a real-time GPU scheduler called TimeGraph [KLRI11], which is based on scheduling policies defined by the user. Each application gets periodically a budget of GPU execution time assigned. Their scheduling is not aware of frames and does not use priorities. Therefore, this approach cannot guarantee a maximum delay for rendering a frame, cannot guarantee certain *frame rate*s, and cannot prioritize. Moreover, due to latency induced by synchronous GPU operations, applications using the X Server and double buffering encounter additional problems addressed in [KLIR11]. However, the X Server itself does not provide sufficient isolation mechanisms and therefore cannot be used for an automotive HMI system.

Yu et al. [YZQ+13] propose a resource management framework called Virtualized GPU Resource Isolation and Scheduling (VGRIS) targeted at cloud gaming systems. This approach introduces a delay after the SwapBuffers command, which represents the finished execution of one frame. It only supports a coarse-grained time resolution since only fully rendered frames are measured and scheduled rather than GPU command groups. They assume that

the rendering behavior of the applications is well known. Thus, they use a cooperative scheduling scheme where applications release the GPU by using SwapBuffers. However, if an application never calls SwapBuffers, it would get infinite GPU execution time. Zheng et al. [ZQCZ16] present GCloud, which extends the concepts of VGRIS to determine the resource consumption of rendering nodes and implement load balancing using simple quality requirements, such as desired *frame rate*s, of the individual games. While GCloud helps to efficiently use cloud systems for a higher number of game instances, its real-time scheduling capabilities are not better than those of VGRIS.

Works like [BK12] provide real-time GPU scheduling for GPGPU, which is actually easier to solve, since the GPGPU frameworks such as CUDA or OpenCL offer much better control of the execution. Therefore, such scheduling concepts work without changing the GPU driver. Unfortunately, they do not support a 3D rendering pipeline of a GPU.

A non-preemptive Highest Priority First (HPF) scheduling (also called *priority scheduling*) [TB14] always schedules the process with the highest priority that is ready. Using it for a Virtualized Automotive Graphics System would meet our goals only for the highest priority application. Lower priority applications could violate deadlines of higher priorities, since HPF is not aware of the deadlines. Furthermore, in terms of GPU utilization, it is sometimes beneficial to schedule a lower-priority application before a high priority application.

The Least Laxity First (LLF) scheduling algorithm proposed by Stewart et al. [SK] calculates the laxity for each process. The laxity is the amount of time by which dispatching can be deferred without violating the deadline. LLF is a preemptive scheduling algorithm that always executes the process with the lowest laxity. In contrast to our scheduling algorithm, LLF does not support priorities, frame-based periodic execution time accounting, and uses preemption. The Maximum-Urgency-First Algorithm (MUF) extends laxity-based scheduling by fixed priorities to support critical tasks. Basically, each fixed priority represents a group of processes in which LLF scheduling is performed. However, assigning each 3D application a fixed priority would be equivalent to HPF scheduling, which does not meet our requirements.

The periodic task model [Liu69, LL73] is a well-known approach to model recurring processes. Other non-preemptive approaches such as the Clairvoyant non-preemptive EDF scheduling [Eke06] are based on it. Each process has a period and a required execution time per period. In our approach, the *frame*

rate implicitly represents a task period and the *etpf* represents the required execution time. However, their task model assumes (for non-preemptive approaches) that the *etpf* represents also the scheduling granularity. This is not true for GPU scheduling, where the granularity of scheduling is CGs, and the *etpf* is just upper bound of the CGs of one frame. Moreover, in our approach the set of tasks, the periods, and the required execution times are all dynamic and priorities are used to select the next CG, if the available GPU resources do not suffice to meet all deadlines.

4.8. Summary and future work

4.8.1. Summary

In this chapter, a GPU scheduling framework for non-preemptive 3D scheduling is presented, providing strong real-time guarantees while still efficiently using the available GPU resources. The GPU command groups submitted by the applications are inserted in dedicated scheduling queues managed by the GPU scheduler thread. Our scheduling algorithm uses application-specific frame deadlines and the estimated execution time of GPU command groups to dispatch commands to the GPU without requiring preemption. Our evaluation on an embedded automotive platform shows that—assuming correct execution time prediction—real-time constraints are guaranteed and a high GPU utilization of 99 % is achieved. In particular, the applications desired *frame rate*s are enforced by the scheduler. If the available GPU resources are not sufficient enough to enforce this for all applications, the applications' priorities determine which command batches are delayed, thus avoiding deadline misses. The execution time of our 3D GPU scheduling algorithm is less than 9 µs on average, thus, the introduced overhead is very low.

4.8.2. Future work

The current scheduling algorithm assumes that the CGs of an application are potentially available at the very beginning of the first period of a frame. That is, the delay between a vsync signal, which lets an application's SwapBuffers call return, and the arrival of the applications first CG of the next frame in the kernel space driver, is expected to be 0 in the best case. However, this assumption is typically too restrictive, since even a very fast application needs some time to submit OpenGL calls and also GPU driver consumes some time before the CG reaches the scheduler. Therefore, we suggest to improve our scheduler by using the minimum submit delay as additional parameter in order to slightly increase the flexibility of the scheduler, which might slightly improve the GPU utilization in some scenarios. Moreover, as motivated in Sec. 4.6, we suggest to adapt our scheduling concept to preemptive embedded GPUs—once they become available.

For automotive systems, the functional safety must be ensured, as described in ISO 26262 [ISO11] (cf., Sec. 1.1.1). Thus, before our scheduling framework could be used for a safety-criticality of ASIL-A or higher, the functional safety must be assessed, which includes—for instance—failures of the GPUs or the connected

displays. While our framework was developed with functional safety aspects in mind, a full certification can only be performed on the whole system (at least the whole vehicle) and is future work. Additionally, software development in the automotive domain is typically affected by OEM-specific requirements, e.g., a development and assessment according to Automotive SPICE [SIG]. To this end, an in-depth code-review of all software components and their implementations needs to be done.

5. Summary

Using 3D graphics in vehicles is becoming more and more popular. In modern high-end cars, the instrument cluster (IC) uses a display to show graphical representations of—traditionally analog—indicators. To this end, 3D rendering is used not only on the head unit (HU), but also for the rendering of critical applications on the IC. At the same time, the trend to use 3D also extends to applications that the driver can install and run—creating the need to also support 3rd-party applications. Traditionally, new functionality in vehicles is often implemented by adding new hardware platforms (ECUs), which provides the required isolation between applications of different safety-relevance. Unfortunately, this practice is costly and disadvantageous concerning energy consumption and installation space. Therefore, all these applications should share a single hardware platform and thus a single GPU.

Based on the relevant ISO standards, automotive design guidelines, legal requirements, and OEM specific demands, this work presented the requirements that apply to automotive application development. It was deduced from these requirements, that isolation for 3D rendering is the major challenge. In this work, we presented a virtualized architecture, and a real-time 3D GPU scheduling concept based on execution time prediction.

We proposed the Virtualized Automotive Graphics System (VAGS), which uses a microkernel-based hypervisor to isolate VMs. This allows to replace the physical isolation (using separate ECUs) by software-based isolation and thus can run on a single ECU. For instance, the HU and IC would run in separate VMs. To add further functionality, further VMs could be added, e.g., for 3rd-party applications. The VMs communicate with each other using an isolated communication channel. The VAGS uses a dedicated Virtualization Manager VM, which has exclusive hardware access and is responsible to provide safe access to the shared resources. In particular, it contains a Permission and Policy Management that performs access control to display areas, input events, and GPU rendering. The GPU Scheduler needs to support priorities and deadlines in order to fulfill the requirements of critical 3D applications, while

maintaining isolation. Since current GPUs cannot be preempted, a non-preemptive scheduling approach is required. This requires that the execution times of the command groups (CGs) are known in advance, i.e., they have to be predicted. Besides, knowing the predicted execution time significantly helps the GPU scheduler to achieve a high GPU utilization.

To this end, we presented a novel framework to measure and predict the execution time of GPU commands using OpenGL ES. For Draw commands we presented two heuristics that estimate the number of fragments, two concepts that estimate the shader execution time, and an optional online adaption mechanism. The number of fragments is estimated either by the bounding box of the rendered model, on which the vertex shader projection is applied, or by a subset of the triangles, which is used to estimate the average size of a triangle. To estimate the shader execution time, we either execute them in a profiling environment with a dedicated OpenGL ES 2.0 Context, or we use a MARS (multivariate adaptive regression splines) model trained with offline data. With our implementation and evaluation of the framework we demonstrated its feasibility and showed that good prediction accuracy can be achieved. For instance, when rendering a popular 3D benchmark scene, less than 0.4 % of the samples were underestimated by more than 100 µs and less than 0.2 % of the samples were overestimated more than 100 µs. We showed that the overhead introduced by our implementation is low and on the long-run sometimes even negligible.

Moreover, we presented a real-time 3D GPU scheduling framework that provides strong guarantees for critical applications while still giving as much GPU resources to less important applications as possible, thus ensuring a high GPU utilization. The scheduler operates fully dynamic, which means that the set of applications and the CGs submitted by them can dynamically change over time. We presented our implementation on an automotive embedded platform with Linux. By our evaluations, we showed that the scheduler is effective in giving applications with higher priority precedence on GPU time resources. In a challenging scenario with 17 applications, it achieved a high GPU utilization of 99 % and met 99.9 % of the deadlines of the highest-priority application. Moreover, we showed that scheduling is performed highly efficient in real-time with less than 9 µs latency, which is only about 10 % of the time the dispatching of a CG by the native GPU driver in kernel space takes.

A. Appendix

A.1. Vivante GPU instruction set

In Listing A.1, the shader instruction set of the Vivante GC2000 GPU is shown. The names were obtained from the debugging symbols of the user space driver.

Listing A.1:

```
 0 NOP
 1 MOV
 2 SAT
 3 DP3
 4 DP4
 5 ABS
 6 JMP
 7 ADD
 8 MUL
 9 RCP
10 SUB
11 KILL
12 TEXLD
13 CALL
14 RET
15 NORM
16 MAX
17 MIN
18 POW
19 RSQ
20 LOG
21 FRAC
22 FLOOR
23 CEIL
24 CROSS
25 TEXLDPROJ
26 TEXBIAS
27 TEXGRAD
28 TEXLOD
29 SIN
30 COS
31 TAN
32 EXP
33 SIGN
34 STEP
35 SQRT
36 ACOS
37 ASIN
38 ATAN
39 SET
40 DSX
41 DSY
42 FWIDTH
43 DIV
44 MOD
45 AND_BITWISE
46 OR_BITWISE
47 XOR_BITWISE
48 NOT_BITWISE
49 LSHIFT
50 RSHIFT
51 ROTATE
52 BITSEL
53 LEADZERO
54 LOAD
55 STORE
56 BARRIER
57 STORE1
58 ATOMADD
59 ATOMSUB
60 ATOMXCHG
61 ATOMCMPXCHG
62 ATOMMIN
63 ATOMMAX
64 ATOMOR
65 ATOMAND
66 ATOMXOR
67 TEXLDPCF
68 TEXLDPCFPROJ
69 —
70 —
71 —
72 —
73 —
74 —
75 —
76 —
77 —
78 —
79 —
80 SINPI
81 COSPI
82 TANPI
83 ADDLO
84 MULLO
85 CONV
86 GETEXP
87 GETMANT
88 MULHI
89 CMP
90 I2F
91 F2I
92 ADDSAT
93 SUBSAT
94 MULSAT
95 DP2
96 UNPACK
97 IMAGE_WR
98 SAMPLER_ADD
99 MOVA
100 IMAGE_RD
101 IMAGE_SAMPLER
102 NORM_MUL
103 NORM_DP2
104 NORM_DP3
105 NORM_DP4
106 PRE_DIV
107 PRE_LOG2
```

A.2. libETP XML profiling data file

Listing A.2 shows an XML settings file of libETP that contains the profiling data for the "es2gears" benchmark application. The "main" method was shortened due to space restrictions.

Listing A.2:

```
<?xml version="1.0" encoding="UTF-8"?>
<Profiling version="15">
   <System gpu="Vivante" kernel="NN">
      <Program Code="619561618">
         <!--attribute vec3 position;
attribute vec3 normal;

uniform mat4 ModelViewProjectionMatrix;
uniform mat4 NormalMatrix;
uniform vec4 LightSourcePosition;
uniform vec4 MaterialColor;

varying vec4 Color;

void main(void)
{
...
}-->
         <!--precision mediump float;
varying vec4 Color;

void main(void)
{
    gl_FragColor = Color;
}-->
         <etp_1V>13241</etp_1V>
         <etp_1F>159</etp_1F>
         <KF_Draw>2.229586</KF_Draw>
         <KF_SB>0.946322</KF_SB>
      </Program>
   </System>
</Profiling>
```

A.3. Additional results for scheduler effectiveness

This appendix section extends the results presented in Section 4.5.2.

A.3.1. Influence of MPCG on scheduler effectiveness

The effectiveness of the GPU scheduling for the setup described in Table 4.1 was evaluated with values for *MPCG* other than 5. In Fig. A.1, the results for `MPCG` = 2 are depicted. A comparison to the results for `MPCG` = 5 (Fig. 4.14)

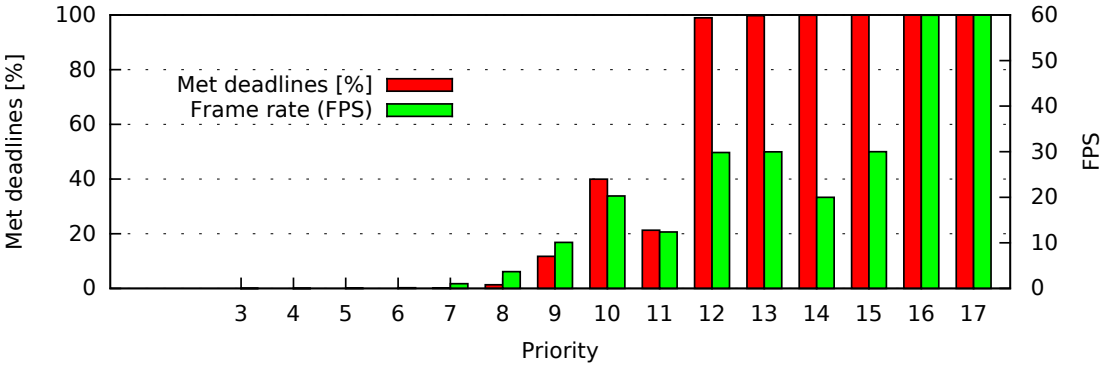


Figure A.1.: Effectiveness (mixed scenario), *MPCG*=2

shows that a lower *MPCG* reduces missed deadlines of the Priorities 12–13. This is due to the fact that less CGs of Quake 3 can be pending—each cumbered with a sometimes high overestimation—resulting occasionally in more deadline misses. Increasing *MPCG*, e.g., to `MPCG` = 10 in Fig. A.2, significantly increases the number deadline misses of the Priorities 16–17 and 12–13, since occasionally the command queue contains multiple pending CGs that were overestimated and the scheduler could not preempt the GPU. A higher *MPCG* potentially increases GPU utilization (cf., Fig. 4.17), but might also result in more missed deadlines, if the execution time prediction underestimates multiple consecutive CGs.

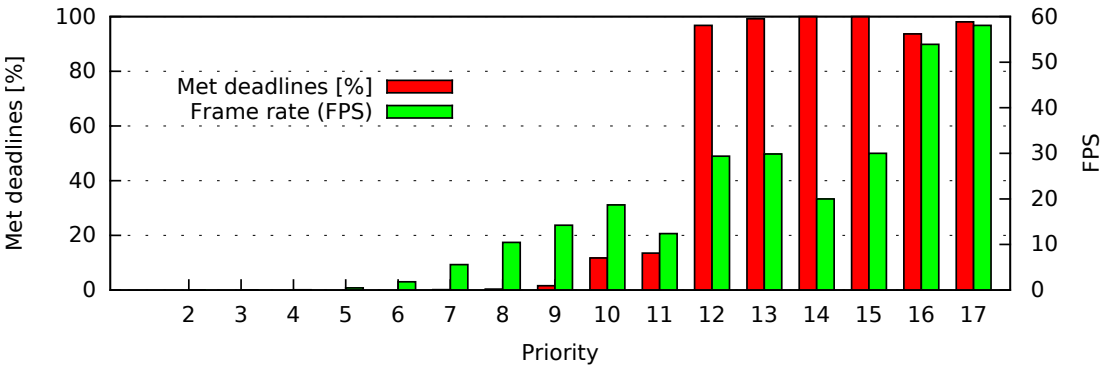


Figure A.2.: Effectiveness (mixed scenario), *MPCG*=10

A.3.2. Scheduler effectiveness with huge ETP error

The effectiveness of the GPU scheduling for the setup described in Table 4.1 was evaluated with additional biased prediction factors for Quake 3. In Fig. A.3, we depict the results if for Quake 3 only 25 % of the execution time of each CG was used. Since CGs with smaller ETP can more often be scheduled, the *frame rate* of Quake 3 increases slightly (compared to Fig. 4.14). However, since all CGs are massively underestimated, the deadline misses of higher-priority applications increase, especially for Priority 12, which is most affected since it is the next higher priority. Nevertheless, even with such a huge prediction error the *frame rate* of the Priorities 16 and 17 are still very close to the desired 60 FPS. In Fig. A.4, we depict the results if all CGs of Quake 3 are predicted with infinite execution time. Thus, Quake 3 cannot be scheduled at all and the Priorities 12–17 always meet their deadlines. This shows that the deadline misses of the Priorities 12–17 observed in the Figures A.3, 4.14, and 4.15, are caused by ETP errors of Quake 3 and not the fault of the scheduler.

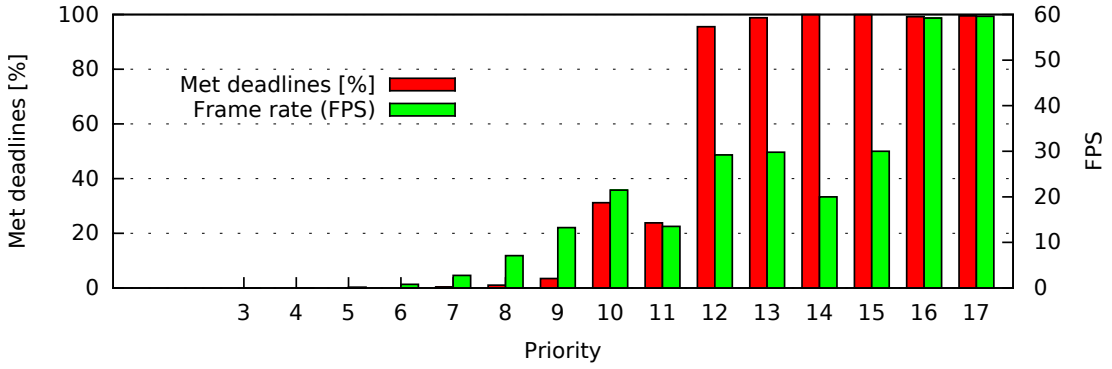


Figure A.3.: Effectiveness (mixed scenario), Quake 3 with 25 % *predET*

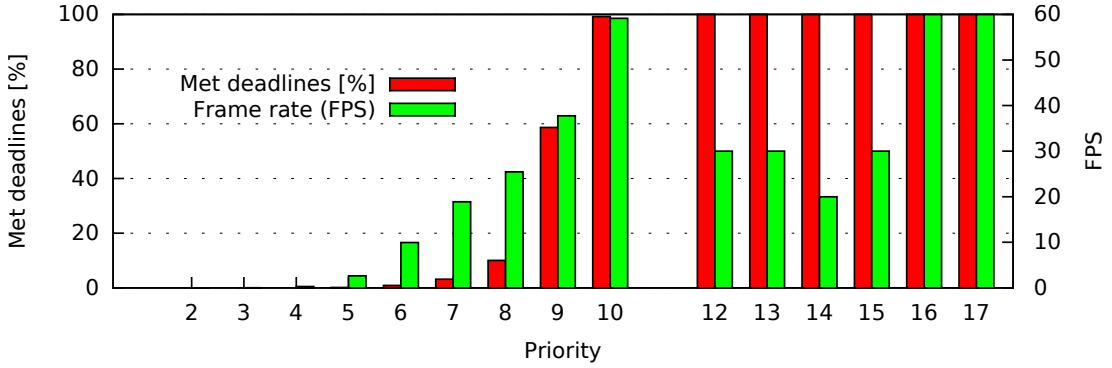


Figure A.4.: Effectiveness (mixed scenario), Quake 3 with $predET = \infty$

A.3.3. Scheduling timing

This section extends the results presented in Section 4.5.2. In Fig. A.5 we depict a timing diagram where a short sequence of the mixed scenario evaluation with `MPCG` = 1 is shown. While P17 denotes the highest priority (a speedometer), P1 denotes the lowest priority (an instance of glmark2). At the lines P17 to P1, it is depicted when the respective application submitted a new CG and how long the estimated execution time was. The activity of the GPU scheduler kernel thread is depicted in the line SCHED. Since the command queue has a size of one, the GPU goes idle after each executed CG.

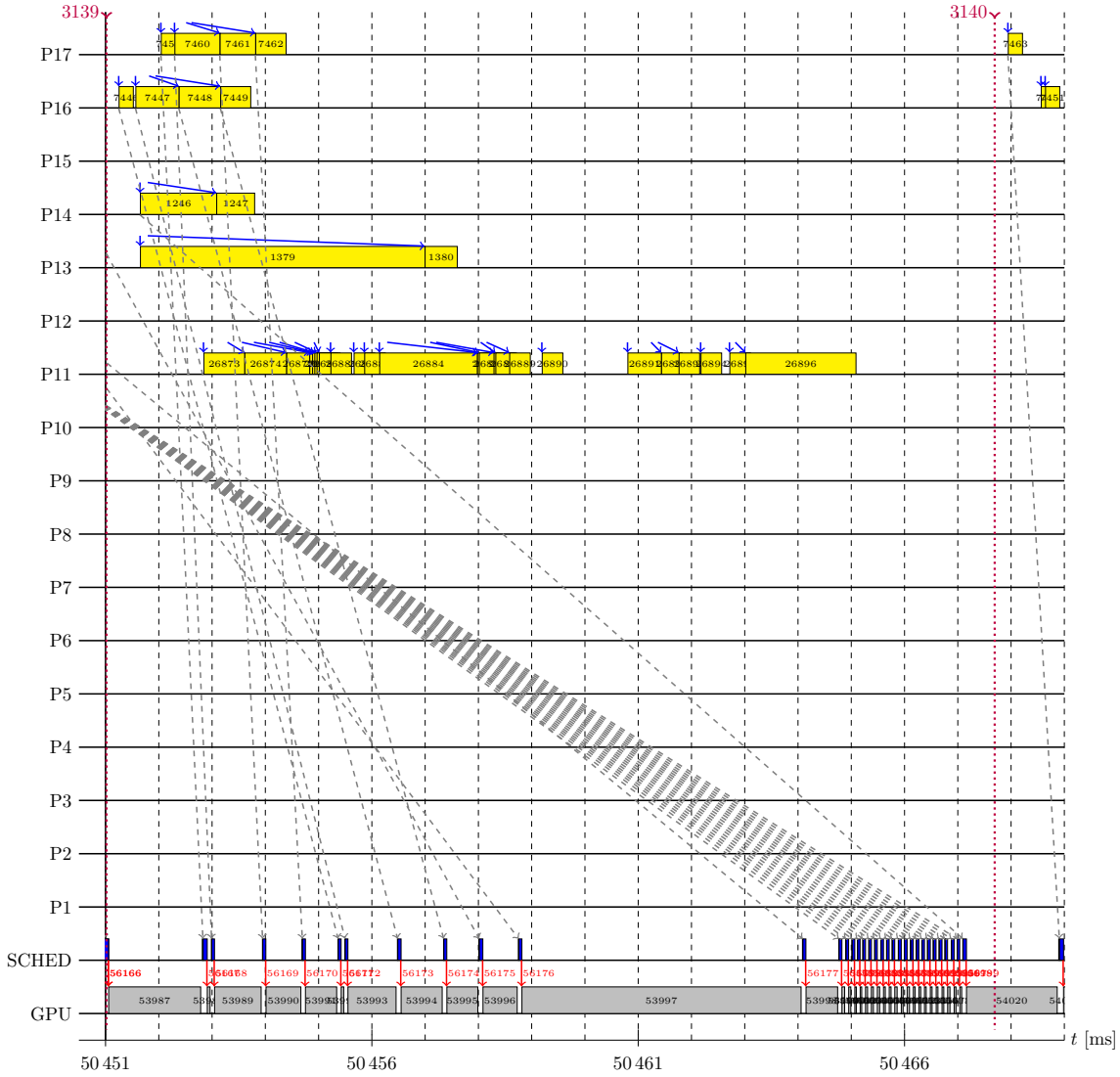


Figure A.5.: Timing diagram of a short period, *MPCG*=1

With *MPCG*>1, the GPU typically does not become idle between different CGs, since the scheduler algorithm and the dispatching can run in parallel to

the GPU execution. There are two possibilities, why the GPU can become idle. First, if the scheduling algorithm could not select a schedulable CG. Second, if the GPU executes a few consecutive CGs faster than the scheduler can dispatch. The second case occurs rarely, especially with *MPCG*>=5. In Fig. A.6, a timing diagram of the evaluation with *MPCG*=5 is depicted. The depicted sequence shows multiple of such situations between $t = 50\,444.3\,\text{ms}$ and $t = 50\,445\,\text{ms}$, where multiple consecutive CGs have almost zero GPU execution time and the GPU becomes idle for a few times, during which the scheduling algorithm runs.

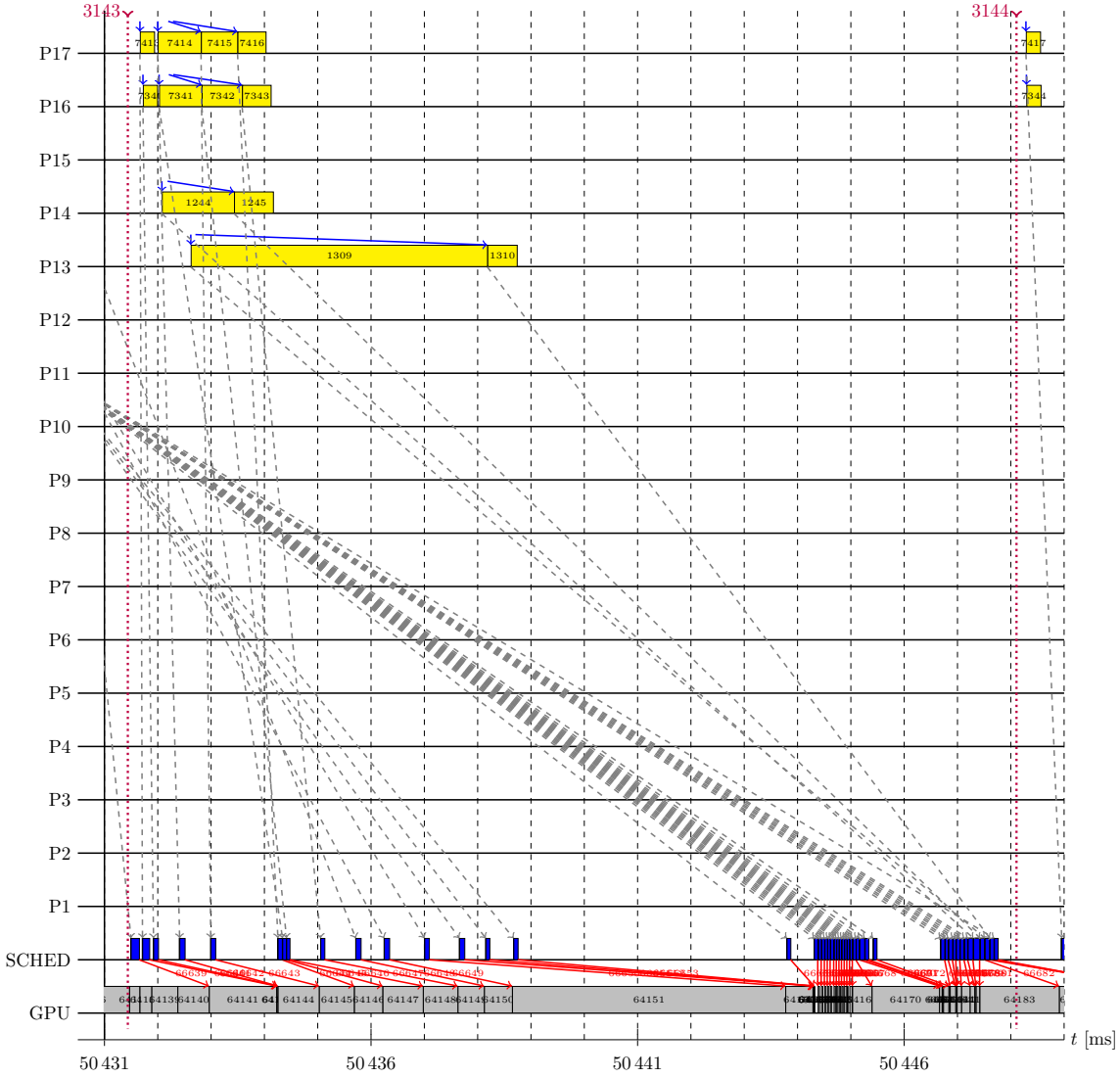


Figure A.6.: Timing diagram of a short period, *MPCG*=5

Glossary

blending

blending is an optional step in the fragment post-processing of the OpenGL rendering pipeline. If enabled, it uses a custom blending function which calculates the new pixel color using the color value of the existing pixel and the color value of the fragment.

buffer resolve

Some GPUs (such as the Vivante GPU used in this work), render in a proprietary format into render buffers. We call the step of converting this format to the target format a buffer resolve. The Vivante driver also uses the term "resolve" for this purpose.

command queue

When a CG is dispatched by the GPU driver, it is inserted into the command queue of the respective GPU. The GPU fetches the CGs from there and executes them.

command buffer

To submit CGs to the GPU driver, an application first connects to the GPU driver. The GPU driver then allocates command buffers and maps them into the application's memory address space. The application inserts GPU instructions into a command buffer and eventually submits them as a CG by using a system call.

command group

A command group is a batch of GPU commands. Typically, this batch is executed by the GPU in one go and thus is the granularity of choice at which GPU scheduling is performed.

correction factor

Our concept for online adaption maintains a correction factor per OpenGL program and type of CG (e.g., for type "DRAW"). The correction factor is applied to the estimated execution time of a CG and continuously updated based on the measured execution times of finished CGs. See Section 3.6.

coverage factor

One of our concepts to estimate the number of fragments is based on a bounding box. Since the actual number of fragments is often smaller than the area covered by the bounding box, we multiply with a configurable coverage factor. See Section 3.5.1.4 and Section 3.7.3

depth test

The depth test is an optional step in the fragment post-processing of the OpenGL rendering pipeline. If enabled, the GPU compares the depth value of each fragment with the depth value of the previously drawn pixel at the target location. The pixel is only updated if the fragment passes the depth test

early depth test

Conceptually, OpenGL performs the depth test after the fragment shader execution. However, in cases where the fragment shader does not change the depth value, the GPU can speed up execution if the depth test is performed early, i.e., before fragment shader execution.

EGL

The EGL API [Lee14] connects rendering (e.g., with OpenGL ES 2.0) with the underlying native windowing system. See Sec. 3.1.1

electronic control unit

An electronic control unit is a hardware component used in modern vehicles. It consists of electronic components, such as microcontrollers, microprocessors, or GPUs. ECUs are typically connected to other ECUs in the vehicle.

etpf

Application-specific parameter used by the scheduler that can be used to increase GPU utilization. It represents an upper bound to the GPU

execution time required to render one frame of the application. See Sec. 4.3.2. In the scheduling algorithm, the *etpf* is represented as attribute of a scheduling queue.

Execution Time Monitor

In our execution time prediction architecture, the Execution Time Monitor measures the GPU execution time for each CG. See Sec. 3.3.3.

fragment shader

A fragment shader is a program written in GLSL. It uses varying variables and uniform variables as input data and calculates the fragment color.

frame rate

3D applications render into a dedicated buffer that is updated at the end of each rendering loop, i.e., when a new frame is rendered. The rate at which this update occurs is called *frame rate* and measured in the unit frames per second (FPS). Additionally, each 3D application has a target *frame rate* that our GPU scheduler tries to fulfill. In our scheduling algorithm, the target *frame rate* is an attribute of the application's scheduling queue.

frames per second

The *frame rate* at which an application updates its render buffer is given in the unit frames per second (FPS).

GPU Profiler

A component of our Predictor that determines the performance of the 3D GPU through profiling and provides it to the prediction models, see Sec. 3.3.2.1.

graphics processing unit

The graphics processing unit (GPU) is an electronic circuit used as hardware accelerator to speed up the graphical rendering. Typically, it has many parallel compute units, caches, and is connected to the system bus.

head unit

The head unit (HU) of a vehicle is responsible for entertainment. Applications running on the HU include radio, TV, navigation directions, web browsing, and configuring comfort functionality.

incipient arrival time

The earliest possible arrival time of a CG, i.e., two *vsync period*s before its desired next deadline.

instrument cluster

The instrument cluster displays the important status of the vehicle, such as speed, fuel level, and critical warnings. Modern high-end ICs use 3D rendering on a display instead of analog indicators.

interrupt service routine

An interrupt service routine is a function within the operating system kernel that is executed when the hardware component it is responsible for (e.g., a graphics processing unit) sent an interrupt.

libETP

Our execution time prediction is implemented as a shared library—called libETP—which intercepts OpenGL ES 2.0 API calls, connects to the Execution Time Monitor, and contains the Predictor.

model view projection matrix

When performing 3D rendering, typically 3D objects and the camera are placed in some orientation into a virtual 3D space and the GPU calculates the respective pixels. The model view projection matrix is a 4x4 matrix that combines object placement (model matrix), camera placement (view matrix), and camera perspective (projection matrix). See Sec. 3.1.2.

OpenGL ES Context Monitor

In our execution time prediction architecture, the OpenGL ES Context Monitor keeps track of the OpenGL ES 2.0 Context, which is required by the Predictor. See Sec. 3.3.1.

OpenGL program

A program object is fully processed shader code consisting of vertex shader and fragment shader which are compiled and linked together. To issue a DRAW call, exactly one Program Object (which is represented by an ID) must be selected.

OpenGL Shading Language

The OpenGL Shading Language is a C-like programming language used for writing the code of shader programs such as vertex shader and fragment shader.

OpenGL rendering pipeline

A sequence of steps performing the actual rendering on the GPU. The major steps are vertex processing, rasterization, fragment processing, and fragment post-processing. See Sec. 3.1.2 and Fig. 3.1.

OpenGL ES 2.0 Context

An OpenGL ES 2.0 Context is associated to an EGL context. It represents a set of OpenGL parameters which can be changed using OpenGL API calls. Some OpenGL API depend on the OpenGL ES 2.0 Context. For instance, whether a glDraw call performs a depth test depends on the current OpenGL state.

OpenGL ES 2.0

OpenGL ES 2.0 is a vendor-independent API for 3D rendering on embedded systems such as smartphones, automotive head units and instrument clusters. It is published by the Khronos Group [ML10].

original equipment manufacturer

OEMs such as car manufactures produce and sell products under their own brand to customers.

Predictor

The main component of our execution time prediction architecture. It provides the execution time for each CG to the GPU Scheduler. See Sec. 3.3.2.

refresh rate

The rate at which a screen (i.e., a display device) updates its content, which is also the rate at which vsync events occur. For TFT displays, a typical refresh rate is 60 Hz.

render target

A render target is a buffer in memory that serves as destination for the OpenGL rendering pipeline and has a defined width and height. A render target is either a render buffer that can be shown on a display or in a window, or it is a texture that serves as input to subsequent DRAW commands.

render buffer

A render buffer is a render target that can be directly shown on a display, or bitblitted by a window manager to a display area.

rendering loop

The rendering loop is a code section typically used by a 3D application, which frequently wants to update the screen and therefore renders one frame after the other. Each loop iteration renders exactly one frame.

reserved

Variable of a scheduling queue that represents the predicted execution times of all command groups of the current frame that must be reserved in order to meet the queue's deadline. Unless all these command groups were already received by the scheduler, the *enqueued*, but at least the *etpf*, is reserved.

scheduling queue

For each thread of an application that submits CGs to the kernel, the GPU scheduler maintains a scheduling queue in kernel space. It contains the scheduling parameters (such as *priority*, *etpf*, and *frame rate*), scheduling state (such as the next deadline), and the sequence of CGs waiting for dispatching.

uniform variable

Uniforms are per-draw input data of shader programs.

varying variable

Varyings are (optional) per-vertex output data of the vertex shader serving as additional input parameters to the fragment shader.

vertex buffer object

A VBO stores vertex array data (vertex attributes). It can provide a performance gain if the data rarely changes, since once the data was written to the VBO it can be used arbitrarily often.

vertex attribute

Attributes are per-vertex input data of the vertex shader

vertex shader

A vertex shader is a program written in GLSL. It uses vertex attributes and uniform variables as input data and calculates the vertex position and varying variables.

Virtualized Automotive Graphics System

A Virtualized Automotive Graphics System is our concept to use virtualization to consolidate the hardware of HMI-related ECUs, in particular the HU and IC. Isolation for 2D and 3D rendering is provided by access control and GPU scheduling concepts.

vsync

Vsync is short for "vertical synchronization". Displays operate at a defined *refresh rate*, at which the screen content is transferred from the GPU to the display image by image. To avoid tearing, the rendering on the GPU is typically synchronized (i.e., vsync) to the short gap after one image was transferred and before the transmission of the next image starts. To allow for synchronization, the GPU driver notifies about each gap using a vsync event.

vsync period

The length of time between two consecutive vsync events (recurrence interval), i.e., $\frac{1}{\texttt{refresh\_rate}}$. In our scheduling algorithm, specific *vsync period*s are distinguished from each other using a sequence number that is denoted by the suffix "#".

Acronyms

CG

command group

ECU

electronic control unit

ETP

execution time prediction

FPS

frames per second

FS

fragment shader

GLSL

OpenGL Shading Language

GPU

graphics processing unit

HU

head unit

IC

instrument cluster

ISR

interrupt service routine

ML

machine learning

MVPM

model view projection matrix

OEM

original equipment manufacturer

VAGS

Virtualized Automotive Graphics System

VBO

vertex buffer object

VS

vertex shader

Math Terms

$\#A_1$

Number of vertex attributes of type float of the OpenGL program

$\#A_2$

Number of vertex attributes of type vec2 of the OpenGL program

$\#A_3$

Number of vertex attributes of type vec3 of the OpenGL program

$\#A_4$

Number of vertex attributes of type vec4 of the OpenGL program

$\#FS_{cmds}$

Array that contains for each GPU instruction the number of calls per fragment shader instance.

$\#VS_{cmds}$

Array that contains for each GPU instruction the number of calls per vertex shader instance.

$\#V_1$

Number of varying variables of type float of the OpenGL program

$\#V_2$

Number of varying variables of type vec2 of the OpenGL program

$\#V_3$

Number of varying variables of type vec3 of the OpenGL program

$\#V_4$

Number of varying variables of type vec4 of the OpenGL program

current#

The sequence number of the current *vsync period*.

desired#

Variable of a scheduling queue that denotes the *vsync period* whose end is the deadline that has to be met if the application's shall get its desired *frame rate*.

dispatched

Variable of a scheduling queue that represents the predicted execution times of all command groups of the current frame that were received (and therefore enqueued) by the scheduler so far.

emulate$_{\triangle}$(UnifiedTriangle)

A function which approximates the size of the projected area of a given triangle. See Sec.3.5.1.2.

enqueued

Variable of a scheduling queue that represents the predicted execution times of all command groups of the current frame that were received (and therefore enqueued) by the scheduler so far.

finish#

After a command group has finished execution, the variable *finish#* of a CG holds the *vsync period* at which its execution was actually finished.

FPLA

Abbreviation for "*vsync period*s look ahead"; a parameter of the scheduling algorithm that determines for how many future *vsync period*s it has to be verified that no higher-priority deadlines are missed.

m_{auxFS}(ctx)

MARS model that estimates the auxiliary time per fragment shader instance, i.e., the time needed to rasterize and load the fragment data for shader execution and perform fragment post-processing such as depth test and blending. See Sections 3.5.3.1 and 3.5.3.3.

$m_{\text{auxVS}}(ctx)$

MARS model that estimates the auxiliary time per vertex shader instance, i.e., the time needed to load the vertex data for shader execution and perform Primitive Assembly. See Sections 3.5.3.1 and 3.5.3.2.

$m_{\text{clear}}(btypes, s_{\text{rb}})$

Our proposed model to estimate the execution time of a CLEAR command. It takes the set of buffer types to clear *btypes* as a bitmask and the size of the render buffer s_{rb} as arguments and is described in Sec. 3.4.2

$m_{\text{cmdsFS}}(ctx)$

MARS model that estimates the time per fragment shader instance required to execute the fragment shader instructions. See Sections 3.5.3.1 and 3.5.3.4.

$m_{\text{cmdsVS}}(ctx)$

MARS model that estimates the time per vertex shader instance required to execute the vertex shader instructions. See Sections 3.5.3.1 and 3.5.3.4.

$m_{\text{draw}}(n_{Calls}, vertices, ctx)$

Our proposed model to estimate the execution time of a DRAW command batch. It takes the number of "Draw" GPU instructions n_{Calls}, the set of vertices, and the current OpenGL ES 2.0 Context as arguments and is described in Sec. 3.5.

$m_{\text{flush}}()$

Our proposed model to estimate the execution time of a FLUSH, which is described in Sec. 3.4.1

$m_{\text{FP}}(ctx)$

Represents our submodel that estimates the execution time of the fragment processing t_{FS} per fragment, using either profiling or machine learning. See Sections 3.5 and 3.5.2.

$m_{\text{nF}}(vertices, ctx)$

A submodel of m_{draw} that estimates the number of fragments using either the triangle samples, or the bounding box approach. See Sec. 3.5.1.

$m_{\text{swapbuffers}}(s_{\text{rb}})$

Our proposed model to estimate the execution time of a buffer resolve command such as SwapBuffers. It takes the size of the render buffer s_{rb} as arguments and is described in Sec. 3.4.3

$m_{\text{texld}}(shader)$

MARS model that estimates the overhead introduced by shader lookup commands in the (fragment or vertex) shader code. See Sections 3.5.3.1 and 3.5.3.6.

$m_{\text{VP}}(ctx)$

Represents our submodel that estimates the execution time of the vertex processing t_{VS} per vertex, using either profiling or machine learning. See Sections 3.5 and 3.5.2.

measuredET

After a command group has finished execution, its measured (i.e., real) execution time is available in the variable *measuredET*, which is provided by the Execution Time Monitor.

MPCG

The maximum number of pending CGs is a parameter of the scheduling algorithm and is the upper limit to the number of unfinished CGs in the command queue.

NOW

The current time at which the scheduling algorithm started to execute its schedule_next() method, or the time at which the CG was inserted into the GPU command queue (Listing 4.3).

numAttrFloats

Number of float variables (i.e., components) of the vertex attributes of the OpenGL program ($\#A_1 + 2 \times \#A_2 + 3 \times \#A_3 + 4 \times \#A_4$)

numAttrs

Number of vertex attributes of the OpenGL program ($\#A_1 + \#A_2 + \#A_3 + \#A_4$)

numVarFloats

Number of float variables (i.e., components) of the varying variables of the OpenGL program ($\#V_1 + 2 \times \#V_2 + 3 \times \#V_3 + 4 \times \#V_4$)

numVars

Number of varying variables of the OpenGL program ($\#V_1 + \#V_2 + \#V_3 + \#V_4$)

pc_{blending}

A constant value that is determined by the GPU Profiler which estimates the overhead per fragment of blending for the DRAW command.

$pc_{\text{clear[btypes]}}$

An array of constants that is determined by the GPU Profiler which estimates the execution time for a CLEAR command. The index *btypes* is a bitmask indicating which buffers shall be cleared.

pc_{depth}

A constant value that is determined by the GPU Profiler which estimates the overhead per fragment of the depth test for the DRAW command.

$pc_{\text{drawconst}}$

A constant value that is determined by the GPU Profiler which estimates the overhead per "draw" GPU instruction in the model for the DRAW command.

pc_{flush}

Constant that is determined by the GPU Profiler which estimates the execution time for a FLUSH command.

$pc_{\textit{swapbuffers}}$

A constant value that is determined by the GPU Profiler which estimates the execution time for a SWAPBUFFERS command.

priority

Each application has a unique priority that represents the importance. The scheduling algorithm gives applications with higher priority precedence on GPU time resources. In our scheduling algorithm, the priority is an attribute of the application's scheduling queue.

$pv_{\text{fragment\_shader}}$

Estimated GPU time per fragment shader instance for a given OpenGL program, determined by the GPU Profiler and used for the DRAW command.

$pv_{\text{vertex\_shader}}$

Estimated GPU time per vertex shader instance for a given OpenGL program, determined by the GPU Profiler and used for the DRAW command.

Q[]

Array of the scheduling algorithm that holds all scheduling queues

res[]

Array of the scheduling algorithm that holds for future periods the amount of GPU time required by applications with *frame rate* = *refresh rate*.

res2[]

Array of the scheduling algorithm that holds for future periods the amount of GPU time required by applications with *frame rate* < *refresh rate*.

s_{rb}

Parameter representing the size of a render or target buffer in pixels.

$SDdelay_C$

Conservative (i.e., upper bound) estimation of the CPU time needed by the scheduling algorithm and the dispatching of a command group by the GPU driver.

$SDdelay_O$

Optimistic (i.e., lower bound) estimation of the CPU time needed by the scheduling algorithm and the dispatching of a command group by the GPU driver.

target#

Variable of a scheduling queue that denotes the *vsync period* whose end is the deadline until which the scheduler plans to finish the scheduling queue's current frame.

Bibliography

[AAM06] AAM. *Statement of Principles, Criteria and Verification Procedures on Driver Interactions with Advanced In-Vehicle Information and Communication Systems*. Alliance of Automotive Manufacturers, July 2006. (Cited on pages 34 and 35.)

[ARA16] BMBF project ARAMiS (Automotive, Railway and Avionics Multicore Systems). `http://www.projekt-aramis.de/`, 2016. (Cited on page 25.)

[AUD14] The New Audi TT Instrument Cluster Created with Rightware Kanzi UI Solution. `http://www.rightware.com/the-new-audi-tt-instrument-cluster-created-with-rightware-kanzi-ui-solution`, 2014. (Cited on pages 18 and 167.)

[AUD15] Audi virtual cockpit – first fully digital dashboard. `http://www.audi.de/de/brand/de/neuwagen/tt/tt-coupe/layer/audi-virtual-cockpit.html`, 2015. (Cited on pages 17 and 18.)

[BDC08] Mikhail Bautin, Ashok Dwarakinath, and Tzicker Chiueh. Graphic Engine Resource Management, 2008. (Cited on pages 24, 130, 161, and 179.)

[BHH00] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, pages 87–95, New York, NY, USA, 2000. ACM. (Cited on page 131.)

[BK12] C. Basaran and Kyoung-Don Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *24th ECRTS*, pages 287–296, July 2012. (Cited on page 180.)

[bmw15] BMW 7 Series Sedan: Driver Assistance. http://www.bmw.com/com/en/newvehicles/7series/sedan/2015/showroom/driver_assistance.html, 2015. (Cited on pages 17 and 19.)

[Bou16] Allen Bourgoyne. BRINGING PASCAL TO PROFESSIONALS. http://on-demand.gputechconf.com/siggraph/2016/presentation//sig1658-allen-bourgoyne-pascal-to-professionals.pdf, 2016. (Cited on page 178.)

[Bro56] Robert G. Brown. *Exponential Smoothing for Predicting Demand*. Arthur D. Little, Massachusetts, 1956. (Cited on page 95.)

[Cec14] Riccardo Cecolin. Compositing Concepts for the Presentation of Graphical Application Windows on Embedded. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, April 2014. (Cited on page 31.)

[Con11] Armin Cont. Analyse der Echtzeitfähigkeit und des Ressourcenverbrauchs von OpenGL ES 2.0. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Dezember 2011. (Cited on page 30.)

[Con15] Continental Engineering Services: Embedded Linux Engineering. http://www.conti-engineering.com/www/download/engineering_services_de_en/themes/download_download_channel_download_channel/fact_sheet_embedded_linux_engineering_en.pdf, 2015. (Cited on page 167.)

[Dai13] Mercedes-Benz integration of iPhone App in A-Class. http://www.iphone-ticker.de/mercedes-benz-iphone-integration-a-klasse-30952/, 2013. (Cited on page 18.)

[Dom12] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012. (Cited on page 55.)

[DS09] Micah Dowty and Jeremy Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009. (Cited on page 48.)

[DWA08] ASHOK DWARAKINATH. A Fair-Share Scheduler for the Graphics Processing Unit. Master's thesis, Stony Brook University, 2008. (Cited on pages 24, 130, and 179.)

[Eis14] Andrej Eisfeld. Entwurf und Analyse von Konzepten zur effizienten Datenübertragung von Grafikrendering-Befehlen auf eingebetteten Systemen. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, September 2014. (Cited on pages 31 and 48.)

[EJ09] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42 –52, April 2009. (Cited on pages 17, 38, and 40.)

[Eke06] C. Ekelin. Clairvoyant non-preemptive EDF scheduling. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 7 pp.–32, 2006. (Cited on page 180.)

[EMP+91] J. Epstein, J. McHugh, R. Pascale, H. Orman, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype B3 trusted X Window System. In *Proceedings of the 7th Annual Computer Security Applications Conference*, pages 44–55, Dec. 1991. (Cited on page 47.)

[ESO08] ESOP. *On safe and efficient in-vehicle information and communication systems: update of the European Statement of Principles on human-machine interface.* Commission of the European Communities, 2008. (Cited on pages 34 and 136.)

[FH03] N. Feske and H. Hartig. DOpE – a window server for real-time and embedded systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 74–77, Dec. 2003. (Cited on page 47.)

[FH05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Computer Security Applications Conference*, pages 85–94, Dec. 2005. (Cited on page 47.)

[For13] Ford. Software development kit (SDK), 2013. (Cited on page 18.)

[Fre14] NXP (formerly Freescale) Surround View & Sense Park Assist System. `http://www.freescale.com/SurroundView`, 2014. (Cited on page 19.)

[Fri91] Jerome H. Friedman. Multivariate Adaptive Regression Splines. *Ann. Statist.*, 19(1):1–67, 03 1991. (Cited on pages 56 and 85.)

[FT09] Henry Burchard Fine and Henry Dallas Thompson. *Coordinate Geometry*. The Macmillan Company, New York, USA, 1909. (Cited on page 71.)

[Gan17] Simon Gansel. *Konzepte und Mechanismen für die Darstellung von sicherheitskritischen Informationen im Fahrzeug*. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, April 2017. (Cited on page 49.)

[GE03] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.*, 3:1157–1182, 03 2003. (Cited on page 87.)

[GGS+09] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM. (Cited on page 48.)

[GH13] Ahmad Gilbeau-Hammoud. Erstellung und Evaluation einer Zugriffskontrolle für die Darstellung grahischer Applikationen im Fahrzeug. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Mai 2013. (Cited on page 31.)

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979. (Cited on page 152.)

[glm] glmark2-es2: OpenGL ES 2.0 benchmark. `https://launchpad.net/glmark2`. (Cited on pages 46, 105, and 110.)

[GM08] Eimear Gallery and Chris J. Mitchell. Trusted Computing: Security and Applications, May 2008. (Cited on page 41.)

[GPSM14] Dan Ginsburg, Budirijanto Purnomo, Dave Shreiner, and Aaftab Munshi. *OpenGL ES 3.0 Programming Guide*. Addison-Wesley, 2nd edition, 2014. (Cited on pages 53, 68, and 73.)

[GSC+15] Simon Gansel, Stephan Schnitzer, Riccardo Cecolin, Frank Dürr, Kurt Rothermel, and Christian Maihöfer. Efficient compositing strategies for automotive HMI systems. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–10, June 2015. (Cited on pages 30 and 66.)

[GSD+13] Simon Gansel, Stephan Schnitzer, Frank Dürr, Kurt Rothermel, and Christian Maihöfer. Towards Virtualization Concepts for Novel Automotive HMI Systems. In *Proceedings of IESS*, IFIP LNCS. Springer Berlin Heidelberg, 2013. (Cited on page 30.)

[GSGH+14] Simon Gansel, Stephan Schnitzer, Ahmad Gilbeau-Hammoud, Viktor Friesen, Frank Dürr, Kurt Rothermel, and Christian Maihöfer. An access control concept for novel automotive HMI systems. In *Proceedings of the 19th SACMAT*, 2014. (Cited on pages 30, 31, 49, and 136.)

[GSGH+15] Simon Gansel, Stephan Schnitzer, Ahmad Gilbeau-Hammoud, Viktor Friesen, Frank Dürr, Kurt Rothermel, Christian Maihöfer, and Ulrich Krämer. Context-aware access control in novel automotive HMI systems. In *Information Systems Security*, volume 9478 of *LNCS*, pages 118–138. Springer, 2015. (Cited on pages 30, 31, 49, and 136.)

[Han07] Jacob G. Hansen. Blink: Advanced Display Multiplexing for Virtualized Applications. In *Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 15–20, 2007. (Cited on page 48.)

[HEB+01] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference*

on Computer graphics and interactive techniques, SIGGRAPH '01, pages 129–140, New York, NY, USA, 2001. ACM. (Cited on page 131.)

[HHN+08] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pages 43:1–43:10, New York, NY, USA, 2008. ACM. (Cited on page 131.)

[Hoh02] Michael Hohmuth. *The Fiasco kernel: System Architecure.* Technical report: TUD-FI02-06-Juli-2002, 2002. (Cited on page 47.)

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction.* Springer, 2 edition, 2009. (Cited on page 55.)

[Int15] INTEGRITY Multivisor: Secure Virtualization Technology. `http://www.ghs.com/download/datasheets/INTEGRITY_Multivisor.pdf`, 2015. (Cited on page 49.)

[Int16] Intel Graphics Virtualization Technology (Intel GVT). `https://01.org/igvt-g`, 2016. (Cited on page 48.)

[ISH98] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 141–150, New York, NY, USA, 1998. ACM. (Cited on page 131.)

[ISO96] ISO 11428. *Ergonomics – Visual danger signals – General requirements, design and testing.* ISO, Geneva, Switzerland, Dec. 1996. (Cited on pages 34, 36, and 39.)

[ISO02] ISO 15005. *Road vehicles – Ergonomic aspects of transport information and control systems – Dialogue management principles and compliance procedures.* ISO, Geneva, Switzerland, July 2002. (Cited on pages 27, 33, 34, 35, 36, 37, and 38.)

[ISO03] ISO 17287. *Road vehicles – Ergonomic aspects of transport information and control systems – Dialogue management principles and compliance procedures.* ISO, Geneva, Switzerland, April 2003. (Cited on page 129.)

[ISO04] ISO 16951. *Road vehicles – Ergonomic aspects of transport information and control systems (TICS) – Procedures for determining priority of on-board messages presented to drivers.* ISO, Geneva, Switzerland, 2004. (Cited on pages 34, 35, 36, and 37.)

[ISO08] ISO 15408-2. *Information technology – Security techniques – Evaluation criteria for IT security – Part 2: Security functional components.* ISO, Geneva, Switzerland, Aug. 2008. (Cited on pages 34, 35, 36, 38, 39, and 47.)

[ISO10] ISO 2575. *Road vehicles – Symbols for controls, indicators and tell-tales.* ISO, Geneva, Switzerland, July 2010. (Cited on pages 34 and 39.)

[ISO11] ISO 26262. *Road vehicles – Functional Safety.* ISO, Geneva, Switzerland, Nov. 2011. (Cited on pages 18, 27, 33, 34, 38, 49, 136, and 182.)

[JAM04] JAMA. *Guideline for In-vehicle Display Systems – Version 3.0.* Japan Automobile Manufacturers Association, Aug. 2004. (Cited on page 34.)

[Jan11] Helmut Janker. *Straßenverkehrsrecht: StVG, StVO, StVZO, Fahrzeug-ZulassungsVO, Fahrerlaubnis-VO, Verkehrszeichen, Bußgeldkatalog.* C.H. Beck, 2011. (Cited on pages 35, 36, and 136.)

[JE91] J Picciotto J Epstein. Trusting X: Issues in building Trusted X window systems—or—what's not trusted about X. In *Proceedings of the 14th National Computer Security Conference*, volume 1. National Institute of Standards and Technology, National Computer Security Center, Oct. 1991. (Cited on page 47.)

[JRR97] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th*

ACM Symposium on Operating Systems Principles, SOSP, New York, USA, 1997. (Cited on page 178.)

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010. (Cited on page 47.)

[Kel16] Robin Keller. Predicting the GPU Execution Time of 3D Rendering Commands using Machine Learning Concepts. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, May 2016. (Cited on page 30.)

[Khra] OpenGL ES 2.0 Standard. `https://www.khronos.org/opengles/2_X/`. (Cited on pages 28 and 60.)

[Khrb] The Khronos Group. `https://www.khronos.org/`. (Cited on pages 21 and 52.)

[Khrc] WebGL – OpenGL ES 2.0 for the Web. `https://www.khronos.org/webgl`. (Cited on pages 21 and 22.)

[Khrd] WebGL Security. `https://www.khronos.org/webgl/security/`. (Cited on page 21.)

[KLIR11] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-Accelerated Windowing Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 191–200, April 2011. (Cited on page 179.)

[KLRI11] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011. USENIX Association. (Cited on pages 22, 24, 65, 98, 102, 129, 130, 131, 161, and 179.)

[KXG12] T. Kai, X. Xin, and C. Guo. The Secure Boot of Embedded System Based on Mobile Trusted Module. In *2012 Second International Conference on Intelligent System Design and Engineering Application*, pages 1331–1334, Jan 2012. (Cited on page 41.)

[LCTSdL07] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. VMM-independent graphics acceleration. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 33–43, New York, NY, USA, 2007. ACM. (Cited on page 47.)

[Lee14] Jon Leech. Khronos Native Platform Graphics Interface – EGL Version 1.5. `https://www.khronos.org/registry/egl/specs/eglspec.1.5.pdf`, August 2014. (Cited on pages 52 and 194.)

[Liu69] C. L. Liu. Scheduling Algorithms for Multiprocessors in a Hard-Real-Time Environment. *JPL Space Programs Summary 37-60 II*, pages 28–31, 1969. (Cited on pages 24 and 180.)

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20:46–61, Jan. 1973. (Cited on pages 24, 178, and 180.)

[Ma14] Hua Ma. Concepts and Metrics for Measurement and Prediction of the Execution Time of GPU Rendering Commands. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, August 2014. (Cited on pages 30 and 133.)

[MB10] John Maindonald and W. John Braun. *Data Analysis and Graphics Using R: An Example-Based Approach*. Cambridge University Press, New York, NY, USA, 3rd edition, 2010. (Cited on page 85.)

[Mer11] Mercedes-Benz F125 Concept. `http://www.pocket-lint.com/news/112047-mercedes-benz-f152-concept-car-video`, 2011. (Cited on pages 17 and 18.)

[mes] The Mesa 3D Graphics Library. `http://www.mesa3d.org`. (Cited on page 105.)

[MGR+14] Bernhard Mencher, Walter Gollin, Ferdinand Reiter, Andreas Glaser, Felix Landhäußer, Klaus Lerchenmüller, Doris Boebel, Michael Hamm, Tilman Spingler, Frank Niewels, Thomas Ehret, Gero Nenninger, Peter Knoll, and Alfred Kuttenberger. *Overview of electrical and electronic systems in the vehicle*, pages 158–160. Springer Fachmedien Wiesbaden, Wiesbaden, 2014. (Cited on page 17.)

[MGS09] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL® ES 2.0 Programming Guide*. Addison-Wesley, 2009. (Cited on pages 73, 76, and 77.)

[Mil11] S. Milborrow. Derived from mda:mars by T. Hastie and R. Tibshirani. *earth: Multivariate Adaptive Regression Splines*, 2011. R package. (Cited on pages 56, 57, and 85.)

[ML10] Aaftab Munshi and Jon Leech. OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification). `https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf`, November 2010. (Cited on pages 52, 71, 75, and 197.)

[Nou] Nouveau project. `http://nouveau.freedesktop.org/wiki/`. (Cited on page 60.)

[Nvi13] Nvidia automotive driving innovation. `http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf`, 2013. (Cited on pages 17 and 18.)

[ODK+00] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 23–32, New York, NY, USA, 2000. ACM. (Cited on page 131.)

[Ope11] OpenGL Tutorial 3: Matrices. `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices`, 2011. (Cited on pages 53 and 68.)

[OSA] OSADL - Open Source Automation Development Lab eG: latency plot of system in rack #b, slot #7 (Phytec/phyFLEX-i.MX6q @996 MHz, Linux 4.4.15-rt23. `https://www.osadl.org/Latency-plot-of-system-in-rack-b-slot.qa-latencyplot-rbs7.0.html?latencies=&showno=&shadow=0&slider=115`. (Cited on page 102.)

[Pik16] PikeOS: Certified RTOS and Hypervisor in Series Production. `https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/sales_collaterals/01_auto/PikeOS_in_Automotive_Series_Production.pdf`, 2016. (Cited on pages 44 and 49.)

[QNX13] New Version of QNX CAR Platform... `http://www.qnx.com/news/pr_5602_1.html`, 2013. (Cited on page 18.)

[RAL+15] Dominik Reinhardt, Daniel Adam, Enno Lubbers, Rakshith Amarnath, Rolf Schneider, Simon Gansel, Stephan Schnitzer, Christian Herber, Timo Sandmann, Hans-Ulrich Michel, Dirk Kaule, Damla Olkun, Matthias Rehm, Jens Harnisch, Andre Richter, Steffen Baehr, Oliver Sander, Juergen Becker, Uwe Baumgarten, and Henrik Theiling. Embedded Virtualization Approaches for Ensuring Safety and Security within E/E Automotive Systems. In *Embedded World Conference*, 2015. (Cited on page 30.)

[RBE99] Kostadis Roussos, Nawaf Bitar, and Robert English. Deterministic Batch Scheduling Without Static Partitioning. In *Proc. of the Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '99/JSSPP '99, pages 220–235, London, UK, 1999. Springer-Verlag. (Cited on page 178.)

[RM00] Paul Read and Mark-Paul Meyer. *Restoration of motion picture film.* Elsevier, New York, 2000. (Cited on page 153.)

[Röm11] Fabian Römhild. Abschätzung des Ressourcenverbrauchs und Analyse der Echtzeitfähigkeit von CUDA- und OpenCL-Befehlen. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, November 2011. (Cited on page 30.)

[RPr17] The R Project for Statistical Computing. `https://www.r-project.org/`, 2017. (Cited on page 85.)

[RS15] Nathan Reed and Dario Sancho. VR Direct: How NVIDIA Technology Is Improving the VR Experience. `https://developer.nvidia.com/sites/default/files/akamai/gameworks/vr/GameWorks_VR_2015_Final_handouts.pdf`, 2015. (Cited on page 178.)

[rt-18] Suite of real-time tests – cyclictest, ... `https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git`, 2018. (Cited on page 102.)

[SGDR14] Stephan Schnitzer, Simon Gansel, Frank Dürr, and Kurt Rothermel. Concepts for execution time prediction of 3D GPU rendering. In *Proc. of 9th IEEE SIES, 2014*, pages 160–169, June 2014. (Cited on pages 22, 30, 60, and 130.)

[SGDR16] Stephan Schnitzer, Simon Gansel, Frank Dürr, and Kurt Rothermel. Real-time scheduling for 3d gpu rendering. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, May 2016. (Cited on page 30.)

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974. (Cited on page 101.)

[SIG] VDA QMC Working Group 13 / Automotive SIG. *Automotive SPICE–Process Reference Model–Process Assessment Model.* version 3.0, revision id 470 edition. (Cited on page 183.)

[Sim09] Robert J. Simpson. The OpenGL ES® Shading Language. `https://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf`, May 2009. (Cited on pages 22 and 53.)

[SK] David Stewart and Pradeep K. Khosla. Real-time scheduling of sensor-based control systems. In *in Real-Time Programming.* (Cited on page 180.)

[SK10] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM. (Cited on page 47.)

[SKBR12] Robert J. Simpson, John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL ES® Shading Language, July 2012. (Cited on page 22.)

[Smi16] Ryan Smith. Preemption Improved: Fine-Grained Preemption for Time-Critical Tasks – The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation. `http://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10`, 2016. (Cited on page 178.)

[Smo09] Christopher Smowton. Secure 3d graphics for virtual machines. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 36–43, 2009. (Cited on page 47.)

[Sta03] D.H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, 2003. (Cited on pages 27, 33, and 38.)

[SVNC04] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium – Volume 13*, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 47.)

[Tan13] Waqas Tanveer. DEVELOPMENT OF GENERIC SCHEDULING CONCEPTS FOR Open GL ES 2.0. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, July 2013. (Cited on page 30.)

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014. (Cited on pages 24 and 180.)

[TDC14] Kun Tian, Yaozu Dong, and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014*

USENIX Annual Technical Conference (USENIX ATC 14), pages 121–132, Philadelphia, PA, June 2014. USENIX Association. (Cited on page 48.)

[Thi12] Martin Thielefeld. Analyse und Evaluation der Ausführungszeit von OpenGL ES 2.0-Befehlen in Abhängigkeit von Parametern und Kontext. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Oktober 2012. (Cited on pages 30 and 133.)

[WDDa] Windows Display Driver Model (WDDM): GPU preemption. `https://docs.microsoft.com/en-us/windows-hardware/drivers/display/gpu-preemption`. (Cited on pages 178 and 179.)

[WDDb] Windows Display Driver Model (WDDM): Timeout Detection and Recovery (TDR). `https://docs.microsoft.com/en-us/windows-hardware/drivers/display/timeout-detection-and-recovery`. (Cited on page 179.)

[WHJ08] S. Wang, X. Huang, and K. M. Junaid. Configuration of Continuous Piecewise-Linear Neural Networks. *IEEE Transactions on Neural Networks*, 19(8):1431–1445, Aug 2008. (Cited on page 85.)

[YSJZ02] Jian Yang, Jiaoying Shi, Zhefan Jin, and Hui Zhang. Design and implementation of a large-scale hybrid distributed graphics system. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 39–49, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. (Cited on page 131.)

[YZQ+13] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proc. of the 22nd HPDC*, NY, USA, 2013. ACM. (Cited on pages 131, 161, and 179.)

[Zeh14] Felix Zehender. Dynamische Ausführung von Positionstransformationen mittels OpenGL ES 2.0-Shaderprogrammen. Study thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, February 2014. (Cited on page 30.)

[Zha15] Han Zhao. Development and Analysis of a Window Manager Concept for Consolidated 3D Rendering on an Embedded Platform. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, July 2015. (Cited on page 31.)

[ZQCZ16] Youhui Zhang, Peng Qu, Jiang Cihang, and Weimin Zheng. A cloud gaming system based on user-level virtualization and its resource scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1239–1252, May 2016. (Cited on page 180.)

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Stephan Schnitzer)