Real-time Scheduling for 3D GPU Rendering

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

***Abstract*—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. In order to save cost, energy, and installation space, all these applications should share a single GPU. GPU sharing brings up the problem of providing real-time guarantees for rendering content of time-sensitive applications like the speedometer. To solve this problem, we present a real-time GPU scheduling framework which 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. Since current GPUs are not preemptible, we use the estimated execution time of each GPU rendering job to make the scheduling decisions. Our evaluations show that our scheduler guarantees given real-time constraints, while achieving a high GPU utilization of 97 %. Moreover, scheduling is performed highly efficient in real-time with less than 10 μs latency.**

Keywords*—Real-time, GPU scheduling, 3D rendering, automotive HMI, embedded systems

I. Introduction

Innovations in cars are mainly driven by electronics and software today [1]. In particular, graphical functions and applications enjoy growing popularity as shown by the increasing number of displays integrated into cars. For instance, the head unit (HU) uses the center console screen to display the navigation system, or displays integrated into the headrests of the front seats to display multimedia content. Another recent trend in modern cars is to replace the analog instruments of the instrument cluster (IC) by digital 3D displays, for instance as shown in the Mercedes Benz F125 prototype car [2].

Although, in the beginning, graphical output was mainly 2D content such as movies or 2D maps, the amount of 3D graphics is steadily increasing [3, 4]. For example, modern navigation systems display 3D city models. Also, the instruments of the vehicle are rendered 3D objects with reflections and shadows to imitate physical instruments as close as possible. Again, a "bird's eye view" with a virtual 3D model of the car and its surroundings supports the driver during parking. To render such complex scenes with high frame rates, graphical processing units (GPUs) are integrated into cars.

Traditionally, each system such as the HU or IC uses dedicated hardware platforms with integrated GPUs for rendering. However, 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.

For different kinds of applications, the OEMs have specific quality requirements. A few examples for 3D applications are listed next, sorted from stringent requirements to soft requirements. 1) Safety relevant IC applications such as displaying instruments [2, 3] or parking assistant – stutter-free, latency-bound, high frame rates. 2) OEM applications like navigation system – decent quality is important, but low latency and high frame rates are less relevant. 3) Third-party software such as applications from an app store [5–7] or a web browser executing WebGL which are no longer quality-assured by the OEM – best effort, using the remaining GPU resources.

Thus, 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. 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.

Providing such real-time guarantees requires GPU scheduling algorithms. Compared to CPU scheduling, GPU scheduling is challenging since currently GPUs do not support preemption. 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, we have proposed models to estimate the execution time of rendering jobs in [8]. In this paper, we utilize these models to design GPU scheduling algorithms considering 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 summary, we make the following contributions: 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 rates 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.

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

II. Related Work

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 ([9, WDDM]), the implementation of this functionality is optional and no maximum delay between preemption request and completion is guaranteed, making them insufficient for the implementation of real-time scheduling. On the other hand, non-preemptive scheduling approaches typically assume a static set of applications and time requirements.

Bautin et al. [10] developed a system called Graphics Engine Resource Manager (GERM) which targets fairness in GPU multitasking, using a priority-based scheduler. However, this approach does not support frame-based deadlines nor GPU execution time reservation.

Kato et al. developed a real-time GPU scheduler called TimeGraph [11] which is based on scheduling policies defined by the user. Each application gets periodically a budget of GPU execution time assigned. The scheduling is not aware of frames Therefore, this approach cannot guarantee an maximum delay for rendering a frame, nor guarantee certain frame rates.

Yu et al. [12] 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.

Works like [13] provide real-time GPU scheduling for GPGPU which is actually easier to solve, since the GPGPU frameworks like 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 the 3D rendering pipelines of GPUs.

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

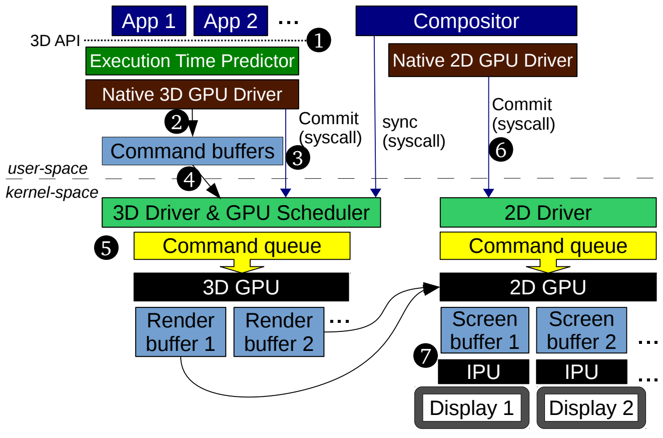

Figure 1. 3D GPU scheduling system model

The *graphic application* (e.g., "App. 1") uses the native GPU drivers for rendering (❶ in Fig. 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 Predictor* predicts the execution time and attaches it to the CG (c.f., [8] for suitable concepts). From the OpenGL commands, the *Native 3D GPU Driver* in user space creates a GPU command batch in a command buffer (❷) and eventually notifies the kernel about it (❸). The kernel-space 3D driver accesses this data (❹) and places 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 screen 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 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 update). After the compositor receives a task, it uses the API calls of the 2D GPU to create a 2D GPU command batch and commit it to the 2D kernel-space driver (❻), which puts it into the 2D GPUs command queue. From there, the 2D GPU fetches and executes the commands which 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). Next, we describe the requirements for 3D GPU scheduling in automotive scenarios.

IV. Requirements

Requirements for rendering in automotive scenarios stem from several sources (cf., [14]):

- Direct legal requirements. Example: as regulated by German law (StVZO §57 [15]), the speedometer must be visible and display the current speed.
- Implicit requirements from standards and automotive guidelines like [16, ISO 26262] and [17]. 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.

It must be noted, that the 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 [18, 19] 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 necessitate 3D GPU scheduling.

From the mentioned requirements, we derive the *priority* as first crucial parameter of each 3D application. The priority is used to guarantee preference to more important or more safety-critical applications. For instance, in an automotive HMI system, the speedometer would get high priority, the navigation system medium priority, and custom third-party applications would get low priority.

As pointed out in Sec. III, the driver notifies about each *vsync event*. A vsync event represents the time when compositing starts. The compositor obviously can only bitblit those frames which have finished before compositing starts, i.e., the vsync event. For the applications, this means that vsync events are actually deadlines for 3D rendering, in order to make the content appear on the display in time. We denote the time interval at which vsync events occur as the *frame period*. As presented in [14] (Requirement “R4.2 – Rendering Time Constraints”), an application has a desired maximum latency between two consecutive frames. Depending on the desired latency, applications therefore have to define an integer multiple – called *framestride* – of the frame period which does not exceed the desired maximum latency. To this end, the framestride is the second crucial parameter of each 3D application. Additionally, the framestride serves as a desired frame rate (measured in FPS, i.e., frames per second). For a screen refresh rate of 60 Hz, a framestride of 1 represents 60 uniformly distributed FPS, a framestride of 2 represents 30 uniformly distributed FPS, a framestride of 3 represents 20 uniformly distributed FPS, etc. However, it is important to note, that a frame rate typically is much less expressive than a framestride. A frame rate just denotes the number of frames finished per second in average. If just a single application is running (which is the typical scenario on today’s end-user systems), frames are usually quite uniformly distributed. However, in a multi-application scenario—as we have in automotive HMI systems—the GPU scheduler is required to ensure that frames are uniformly distributed.

The goal of our scheduling algorithm is, to guarantee the desired framestride for as many of the highest priority applications as possible. Or, in other words, to schedule the CG with the smallest possible priority, such that no deadline of any higher priority application can be violated hereby. Additionally, the GPU utilization shall be as high as possible, i.e., the amount of time the GPU is idle shall be very low. This goal ensures the strict requirements of high priority applications can be met, which well-matches the automotive requirements for 3D scheduling (cf., [14]).

V. Scheduling Concepts

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

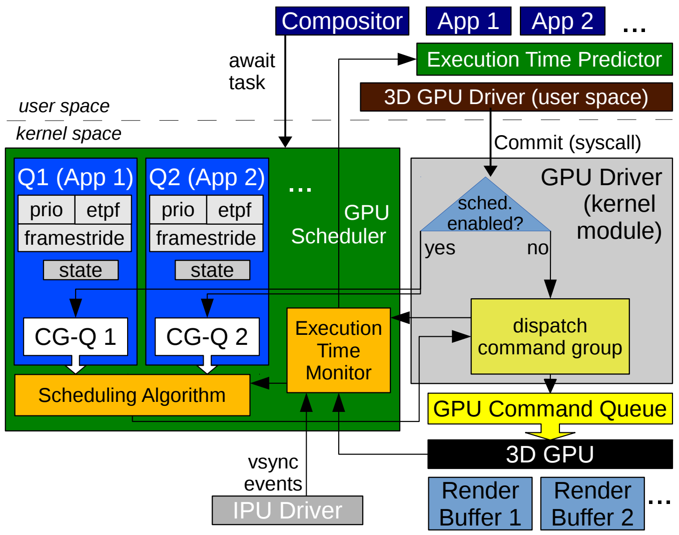

Figure 2. GPU scheduling architecture

A. System Architecture

Our architecture is presented in Fig. 2. As discussed earlier, non-preemptive real-time 3D GPU scheduling requires the execution times of GPU commands to be known in advance. The *Execution Time Prediction* estimates the execution time of each command group (CG) in user space in a shared library located between the user space program and the *3D GPU driver* in the user space. Its implementation uses the concepts of [8]. The *3D GPU driver* in user space commits the CGs to the kernel space driver module where they are dispatched within the system call execution. Dispatching means that the command and associated data from the user space process are verified and inserted into the GPU 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. If the *GPU Scheduler* is enabled (i.e., by loading the scheduler kernel module), it registers with the GPU Driver kernel module. This changes the driver behavior such that commands are no longer directly dispatched but forwarded to the GPU scheduler module instead. For each application, the GPU scheduler maintains a queue for managing commands. The *Scheduling Algorithm* uses these queues, as well as the parameters, such as $priority$, $framestride$, the timestamps of the vsync events, and the internal state (e.g., the next target deadline) to calculate the next CG to dispatch. Eventually, the selected CG gets dispatched using the dispatch function of the GPU Driver. The *Execution Time Monitor* keeps track of all the relevant timestamps, e.g., when a command was inserted, when it was submitted to the GPU, and when it has finished execution. These timestamps are used by the scheduler to trigger the Scheduling Algorithm to submit new commands but also for accounting of already consumed GPU resources.

B. Scheduling Algorithm

As mentioned in Sec. IV each application has a $priority$ and $framestride$. 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 command to get dispatched too late since the dispatched low-priority command cannot be preempted. Such a delayed submission can be due to unfortunate CPU scheduling or different CPU execution time of the application code. Moreover, each time an application enqueues a swapbuffers command, it gets 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. 3 a possible situation where both aspects can be observed. The Lines P2 and P1 depict the enqueued CGs, in which estimtated 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 two command

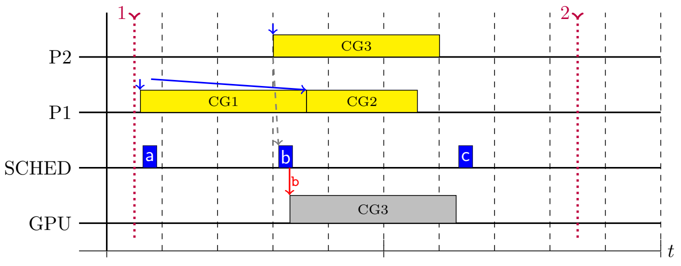

Figure 3. Example for simple priority-based scheduling

groups 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 providing strong guarantees to high-priority applications is maintained, 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

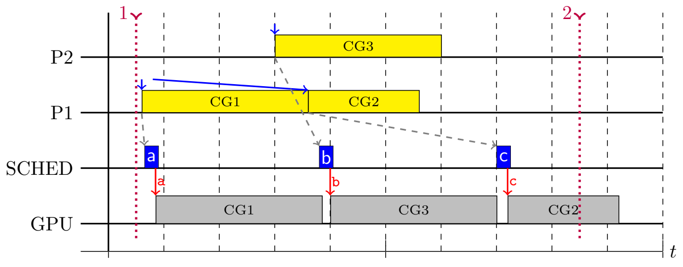

Figure 4. Example for scheduling using $etpf$

utilization (i.e., reduce the GPU idle time), 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 $etpf$ 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 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 impacts 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 we depict the same sequence of received commands as in Fig. 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 end 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 $framestride$ (screen refresh rate divided by $framestride$ represents the number of uniformly distributed frames per second)
- Demanded execution time per frame, called $etpf$

Next, we explain the scheduling which consists of four steps: 1) Command submission by applications. 2) Update the scheduling data (Lines 1 to 6). 3) Select the CG with the lowest possible $priority$ (Lines 7 to 33). 4) Dispatch the selected CG (Lines 34 to 39). The first step—depicted in Listing 1—

is executed by an application thread. The *Scheduling Algorithm*—depicted in Listing 2—is executed by the scheduler kernel thread and consists of Steps 2 through 4. Each step is explained in the following. Variables representing *vsync event* numbers have "#" as a suffix.

1) Command submission by applications: In Listing 1 we depict the code executed if an application submits a new CG.

Listing 1. **submit_command(CG)**

```
1 CG.Q.enqueued += CG.predET
2 if CG.is_swapbuffers
3     CG.Q.res = CG.Q.enqueued
4     sleep until next frame period begins
5 else
6     CG.Q.res = max(CG.Q.etpf, CG.Q.enqueued)
```

In Line 1 we increment the aggregated execution time by the CG's predicted execution time. If the CG contains a swapbuffers command, 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 frame period begins (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. The corresponding vsync event is the same as the one calculated in Line 5 of Listing 2. If the CG does not contain a swapbuffers command, at least the $etpf$ of the CGs queue Q is used as reserved time *res* (Line 6).

Listing 2. **schedule_next** (selects and dispatches next CG)

```
 1 for each CG finished in the meanwhile:
 2     busyUntil += CG.measuredET - CG.predET
 3     if CG.is_swapbuffers
 4         CG.Q.target# = CG.Q.framestride +
 5                     max(CG.desired#, CG.finished#)
 6         CG.Q.desired# = CG.Q.target#
 7 tRef = max(busyUntil, NOW) + SDdelay - t(current#)
 8 res[] = Qdis = 0
 9 for p = n TO 1:
10     if Q[p] not empty:
11         accounted = tRef + Q[p].predET
12         can_schedule = true
13         for i = current# TO (current# + VSM):
14             accounted += res[i]
15             if ((fPeriod * (i + 1)) < accounted):
16                     AND (res[i] > 0):
17                 can_schedule = false
18                 break
19         if can_schedule:
20             Qdis = Q[p]
21         else
22             if (Qdis = 0) AND (res[current#] = 0):
23                 for ps = (p - 1) TO 1:
24                     if Q[ps].predET < (fPeriod - tRef)
25                         Qdis = Q[ps]
26                         break
27             break
28     Q[ps].target# = max(Q[ps].target#,
29                         current# + (tRef / fPeriod))
30     res[Q[ps].target#] += Q[ps].res - Q[ps].disp
31     for i = (Q[ps].target# + Q[ps].framestride)
32          TO (current# + VSM) STEP Q[ps].framestride:
33         res[i] += Q[ps].etpf
34 if Qdis > 0:
35     dispatch(Qdis)
36     busyUntil = max(busyUntil, NOW) + Qdis.predET
37     if Qdis.is_swapbuffers:
38         Qdis.dispatched = 0
39         Qdis.res = Qdis.etpf
40         Qdis.target# += Qdis.framestride
```

2) Update the scheduling data: When the scheduler selects the CG to dispatch, it uses the predicted execution times to calculate until when the GPU will be busy (Line 36) and until which vsync period the CG finishes (Line 40). After the GPU, which executes the CGs concurrently, has finished execution of some CGs, the scheduler uses the measured execution time to update those values. First, the scheduling algorithm updates scheduling times using the finish times of command groups (CGs) that have finished since the last run of the scheduler (Line 1). The variable *busyUntil* holds the timestamp at which the GPU will become idle. Using the measured execution time of the CG, we correct a possible prediction error in *busyUntil* (Line 2). If the finished CG was a swapbuffers command, which means that the respective process just completed a frame, and the application did not meet its desired deadline, its next deadline is deferred such that the desired $framestride$ is the distance between the finished and the next frame (Lines 4 to 6). The desired vsync event (*desired#*, Line 6) is used for monitoring and evaluating the correctness of the algorithm, since *desired#<target#* implies a missed deadline.

3) Select the CG with the lowest possible $priority$: Our scheduling algorithm guarantees that it will never dispatch a CG that could violate a deadline of any other application which has higher priority. In order to provide lower-priority applications as much GPU resources as safely possible, it selects the lowest possible priority which can be scheduled safely, i.e., without violating higher-priority deadlines. This ensures, that low-priority applications also get GPU resources, while higher-priority applications scheduled later still finish in time. The variable *tRef* (Line 7) holds 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. We perform time calculations relative to the start of the current vsync period *time(current#)*. The delay of the scheduling algorithm itself plus the delay introduced by dispatching are denoted by *SDdelay* and postpones *tRef*. In Line 9 we iterate through all available scheduling queues (cf., Fig. 2 on the left), ordered from highest to lowest priority. If a queue *Q[p]* is not empty (Line 10), it might be a candidate for dispatching. The variable *accounted* (Line 11) holds the time at which the GPU would finish when dispatching *Q[p]*.

Next, we calculate *accounted* (Line 11), that is, the time at which this candidate is expected to finish. We initially set *can_schedule* to *true* and then check in the for-loop starting at Line 13 whether reservations of higher-priority applications prohibit dispatching *Q[p]*, in which case we set *can_schedule* to *false* (Line 17) and break the loop. In order to limit the number of vsync periods we have to look into the future, we define *VSM* as the maximum value allowed for the $framestride$. In Line 14 we add for each vsync period *i* the reserved execution times of all higher-priority applications which must be finished before the end of period *i*. If the available time is smaller than the accounted time for any period i, where *res[i]*>0, the candidate is not scheduled, since at least one deadline of a higher-priority application could be missed then. However, if no reservation was made (i.e., *res[i]=0*), dispatching *Q[p]* may extend beyond the respective deadline without violating deadlines of higher-priorities. The available time is calculated by the number of periods multiplied by the duration of a period, i.e., *fPeriod * (i+1)* (Line 15). If Q[p] can be scheduled, it is preserved in *Qdis* (Line 20). If Q[p] cannot be scheduled, typically a schedulable higher-priority applications exists, i.e., *Qdis* > 0, and we leave the main loop (Line 27). However, if no schedulable application was found and no time is reserved for the current period (Line 22), instead of letting the GPU idle, we check if the remaining time of the current period is long enough to execute a lower-priority app (Line 25) [1].

Our calculation of the table of reserved time per period (*res*) guarantees that we never schedule a CG which violates a higher-priority deadline. However, a lower-priority is not guaranteed to be scheduled in every situation where it does not provoke a higher-priority deadline miss, although, in most situations, the algorithm will schedule it. To provide a guaranteed scheduling whenever possible, the full sequence of future CGs must be known to the scheduler, which is unfortunately not feasible, since only the execution time of already submitted CGs is available. In the Lines 28 and 29, we check if the current value of the target deadline sequence number (*Q[p].target#*) still can be met. If not, it is postponed to the earliest possible deadline in the future, which is the current sequence number plus the time *tRef* represented as number of periods.

In the Lines 30 to 33 we merge the execution times of the current queue Q[p] into *res* to reserve the requirements and prevent that scheduling one of the lower-priority queues checked next can result in a deadline miss of Q[p]. For the current frame we reserve the *res* time minus the *disp* time (Line 30). The *res* field holds the reserved execution time, i.e., the aggregated execution time of the commands of the current frame. If the application did not yet enqueue a swapbuffers command, at least its eptf is reserved (cf., Listing 1, Line 6). The variable *disp* holds the aggregated execution time of CGs already dispatched for the current frame. For all future frames, with the $framestride$ as step size, we use the $etpf$ value of the respective queue, as reserved time, as explained earlier in this section.

4) Dispatch the selected CG: Finally, in the Lines 34 to 39, we dispatch the selected CG. We call the dispatch function of the GPU driver which provides us with the time at which the CG was actually submitted to the GPU. This time is used to set *busyUntil* to the expected finish time of the dispatched CG (Line 36). If the dispatched commands contains a swapbuffers command (Line 37), we additionally reset the *res* and *disp* execution times (Lines 38+39) and increment the target deadline. Thus, the next time the algorithm executes, the $etpf$ of Qdis is correctly accounted for the next target deadline.

Since the algorithm depends on assumed execution times, the correct behavior obviously depends on the correctness of those values. In particular, the *SDdelay* must be chosen well, i.e., using the upper bound. Additionally, the predicted execution time *predET* must not underestimate the real execution time. Moreover, the size of the GPU command queue, i.e., the maximum number of pending CGs (MPCG), is an important parameter, which has to be considered for *predET*. For $MPCG = 1$, the GPU is idle while the scheduler runs or dispatches, i.e., for up to *SDdelay* of time. Therefore, the assumed execution time on the GPU plus *SDdelay* is used for *predET*. For $MPCG > 1$, since we do not know the future sequence of dispatching in advance, we assume the worst case for each CG. To this end, we calculate *predET* = MAX(SDdelay / assumed_et).

[1]We check only the current period, since we assume that in typical scenarios at least one higher-priority application has $etpf > 0$ and $framestride = 1$, which implies $res[current\# + 1] > 0$. In the unlikely case that this assumption is wrong, the algorithm can be improved by additionally considering future periods without reserved time.

VI. Implementation

In this section we describe the interface our scheduler uses to interact with the native driver, and the compositor interface.

A. Hardware platform

We implemented the GPU scheduling concept described in Sec. V for a Freescale i.MX6 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 Linux 3.14.

B. 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 which 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 driver directly dispatches those commands while in system call context synchronously. This means, that after having submitted a CG, the application is blocked until the CG has been dispatched. Simple scheduling approaches like [10–12] delay this dispatching individually for the respective processes. However, such a synchronous scheduling is inherently not aware of more than only one undispatched CG, whereas each frame is rendered by multiple CCs. 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 all but the last (i.e., the swapbuffers) CG has been dispatched. Therefore, the execution time needed to render the whole frame could only be estimated by the $etpf$. 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 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 those queues. 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. If 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[2]. 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 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.

C. 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.

The applications submit their CGs accompanied by the expected execution times. As mentioned at the end of Sec. V-B, it is crucial for correct scheduling, that the predicted execution time is accurate. Therefore, our implementation adds a small percentage linearly to the predicted execution time as a safety margin to compensate for slightly underestimated predictions. As explained earlier, we calculate the effective predicted execution time considering the delay of scheduling and dispatch *SDdelay*, and the MPCG (maximum pending CGs).

[2]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.

D. 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. On module load, the scheduler initially calls *mxc_gpusched_register*, 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, the scheduler module can be unloaded and then calls *mxc_gpusched_unregister* to notify the driver that the GPU scheduler mode should be disabled. As depicted in Fig. 2, after the GPU scheduler module is loaded, the driver forwards all new CGs received by applications directly to the scheduler instead of dispatching them. The GPU scheduler is then called by the driver at multiple occasions which are, grouped by origin, depicted in Fig. 5. For each type of occasion, the scheduler module implements a dedicated callback method. An application which wants to access the

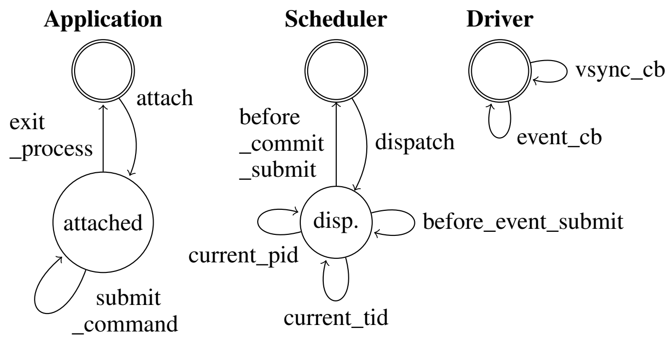

Figure 5. Scheduler interface callbacks

GPU for the first time, makes a system call which attaches 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 calls *submit_command*. If 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 which will be 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*).

E. 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 which 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 simple full compositing approach to demonstrate that compositing easily integrates with our GPU scheduler.

F. 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. 6 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 which submits a new CG

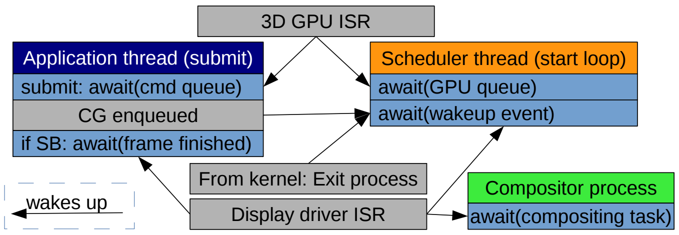

Figure 6. Scheduler thread concurrency synchronization

might be blocked if its associated CG queue (CG-Q), which holds 64 entries, is full. After the GPU has finished an entry, it potentially wakes up the waiting thread. The size of the global GPU queue (*MPCG*) limits the number of pending CGs. The scheduler thread runs in a loop where it first waits until the GPU queue has free slots, then it waits for a wake-up event and eventually executes the algorithm described in Sec. V-B. 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 frame period begins (cf., Listing 1, 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 which protects each access to one of the drivers context structs. Contexts are accessed if 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 which dispatches or executes a detach command. In our algorithm, this would increase the delay of scheduler and dispatching (*SDdelay* in Line 7 of Listing 2) by an relatively high and hard to estimate amount of time. We therefore decided to disable this mutex complex completely while in scheduler mode. This is possible, since attaching creates a new context which cannot cause a conflicting access. All other places are just executed 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 if 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 which 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 which uses a modified execution sequence and one read and one write memory barrier.

VII. Evaluation

In this section, we evaluate the effectiveness, the achieved GPU utilization, and the efficiency of our scheduling approach.

A. Setup

As hardware platform for our evaluations we used a Freescale 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. We used a Yocto 1.8 system image based on Linux kernel 3.14.28 with preempt-rt patches and the recent Vivante driver V5.0.11.p4.25762. As *SDdelay* (delay of scheduler and dispatcher) we used 150 µs. To prevent underestimating the execution time, the predicted execution time values include a safety-margin of 5 %. By default, we used $MPCG = 2$ (i.e., up to two pending GPU commands were allowed), $VSM = 6$ (i.e., the maximum allowed framestride), and evaluated each scenario for 240 s. In the presented results we skipped the first 30 s, since the applications rarely submitted CGs while initializing. The priorities of the daemon threads of the native kernel driver were increased from 0 to 40. The interrupt service routine 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. All evaluated applications were using OpenGL ES 2.0 and EGL as graphics APIs. Since power-management of CPU and GPUs potentially increases latency, it was disabled.

B. Effectiveness

In this subsection, we evaluate the effectiveness of our scheduling approach by showing that 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

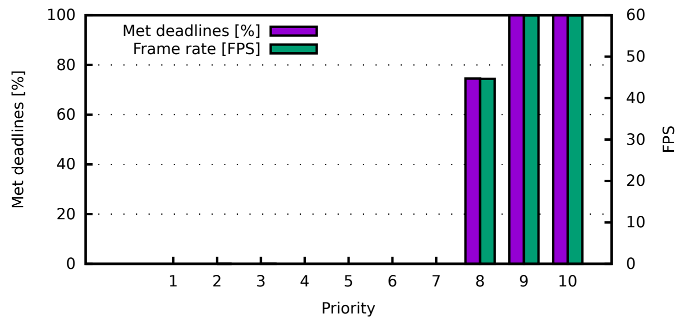

Figure 7. Effectiveness (homogeneous scenario), Framestride=1

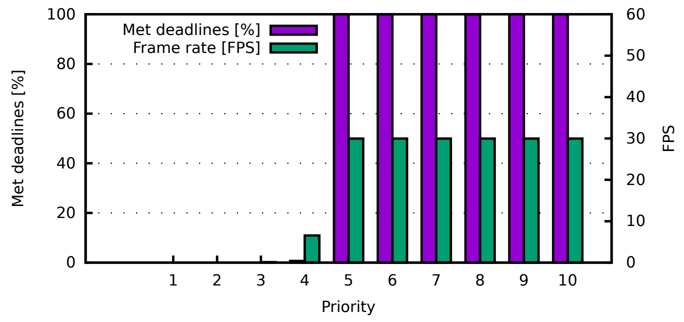

Figure 8. Effectiveness (homogeneous scenario), Framestride=2

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 $etpf = 5\,\mathrm{ms}$. In Fig. 7 we depict the results if each application has $framestride = 1$. For each priority (corresponding to an application instance), we show the percentage of met deadlines and the achieved frame rate. We observe that the deadlines of the priorities 10 and 9 are always met with the requested framestride of 1, which implies a framerate of 60 FPS. Priority 8 met its deadlines in about $75\,\%$ of the cases where its CGs interleaved into the CGs of the higher priority applications. Priorities lower than 7 were never admitted due to the high amount of execution time reserved for higher-priority applications. We additionally evaluated the same scenario using $framestride = 2$ for each application, cf., Fig. 8. In this case, we observe the expected effect that the framerates drop to 30 FPS, and 6 applications (priorities 10 to 5) met all deadlines, since the GPU scheduler now interleaves the CGs in intervals of $33.3\,\mathrm{ms}$ (2 periods) instead of $16.6\,\mathrm{ms}$ (1 period). Thus, the number of applications which fully met deadlines more than doubles compared to $framestride = 1$. The scheduler effectively improves interleaving of the CGs and admits more CGs. Additionally, if all applications have $framestride = 1$, during a short period of time directly after a new vsync period starts, all queues are empty, thus making the GPU idle (about 0.5 to $1.0\,\mathrm{ms}$ each period).

We also evaluated a more sophisticated, mixed scenario, consisting of the set of applications and parameters depicted in Table I. In Fig. 9 we depict the summarized results. We observe that the priorities 9 to 4 almost always meet their deadlines. The few missed deadlines of priorities 9 and 8 are caused by the Execution Time Predictor which often underestimated the CGs of Quake 3, by a couple of milliseconds. While this is not the fault of the scheduling algorithm, it makes clear, that accurate execution time prediction is important. The priority 7 did not suffer, since its deadlines were later, due to higher $framestride$. The priority 6 met its deadlines at about $99.9\,\%$ of its CGs, since $framestride = 3$ always left enough margin to its deadlines to cope with the high prediction errors of Quake 3. In contrast to the priorities 7 and 6, the priorities 5 and 4 suffered additionally from $etpf = 0$, since for future frames no execution time was reserved for them. Priorities 2 and 1 still got GPU resources, namely the resources which could not be used by Quake 3 (priority 3), since the long running or overestimated CGs of Quake 3 would violate deadlines of the priorities 9 to 4. In those cases, the remaining time of a period was used for the lowest priorities 2 and 1, as explained in (cf., Line 22-25 of Listing 2). The embedded GPU is too slow for Quake 3, since even if running stand-alone, Quake 3 achieves only about 20 FPS. Quake 3 therefore gets a fair share of the GPU resources and can render at $11.4\,\mathrm{FPS}$.

Table I. APPLICATION SETUP FOR MIXED SCENARIO

Prio	Framestride	ETPF	Application type	Resolution
9	1	2.4 ms	Automotive speedometer	456x456
8	1	2.4 ms	Automotive tachometer	456x456
7	2	2.3 ms	Glmark2-es2 “shading”	720x540
6	3	2.5 ms	Glmark2-es2 “texture”	720x540
5+4	2	0 ms	Glmark2-es2 “build” (“bunny”)	720x540
3	2	0 ms	Quake 3 demo “four”	1440x540
1+2	1	0 ms	es2gears demo	1280x720

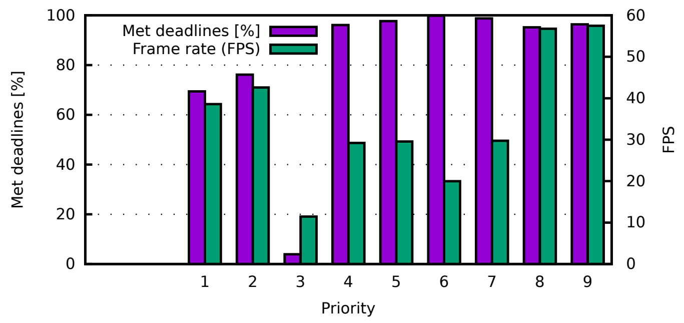

Figure 9. Effectiveness (mixed scenario)

C. GPU Utilization

Next, 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 I. We evaluate the maximum number of pending GPU commands ($MPCG$). The results are depicted in Fig. 10. With a $MPCG = 1$, the average utilization is about $93\,\%$. This is due to the fact, that after the GPU has finished executing a command group, it stays idle while the scheduler selects and dispatches the next command group. This effect gets almost eliminated using $MPCG = 2$, where the utilization is about $97.2\,\%$ on average, since most CGs run longer than *SDdelay* ($150\,\mu\mathrm{s}$) and the next CG is submitted to the GPU before the current one is finished. However, increasing to $MPCG = 3$ shows no further improvement. Since higher values of $MPCG$ accumulate errors of the execution time prediction, $MPCG = 2$ is the best value for our scenarios.

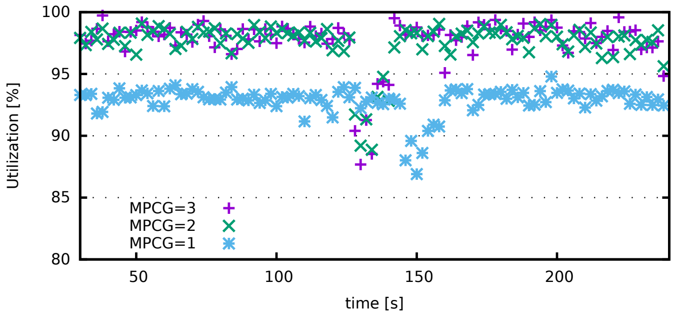

Figure 10. GPU utilization with seven applications

D. 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 1024x768, $framestride = 2$, and $etpf = 1.5\,\text{ms}$. In Fig. 11, we show the average execution time of the 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 algorithm stayed below 10 µs and never exceeded 110 µs. In contrast, the execution time needed to dispatch a CG increased on average from about 50 µs for a single application to about 90 µs for 20 applications. Thus, we conclude that the CPU overhead introduced by our scheduler is small.

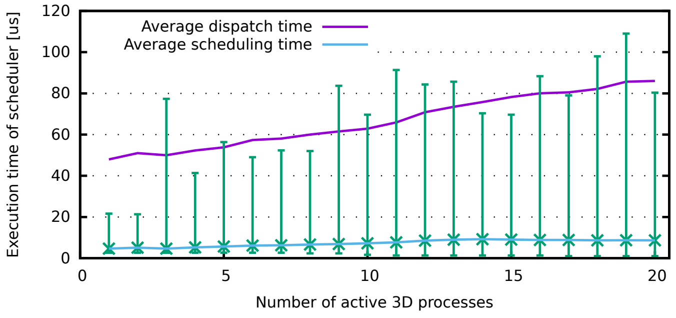

Figure 11. Delay of the scheduling algorithm

VIII. Summary and Future Work

In modern cars, 3D graphical functions enjoy growing popularity. In order to save cost, energy, and installation space, applications of different criticality should share a single GPU. This requires a GPU scheduler which provides real-time guarantees for time-sensitive applications like speedometer and tachometer. In this paper, we presented a GPU scheduling framework for non-preemptive 3D scheduling, providing strong real-time guarantees while still efficiently using the available GPU resources. 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 about 97 % is achieved. The execution time of the scheduling algorithm is less than 10 µs on average, thus, the introduced overhead is low. As part of future work, we want to improve the accuracy of the execution time prediction.

Acknowledgment

This paper has been supported in part by the ARAMiS project of the German Federal Ministry for Education and Research with funding ID 01IS11035.

References

[1] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, 2009.

[2] (2011) Mercedes-Benz F125 Concept. [Online]. Available: http://www.pocket-lint.com/news/112047-mercedes-benz-f152-concept-car-video

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

[4] (2014) Audi R8 Concept Car. [Online]. Available: http://www.audi.com/content/com/brand/en/vorsprung_durch_technik/content/2014/01/showcar-ces.html

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

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

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

[8] S. Schnitzer *et al.*, "Concepts for execution time prediction of 3D GPU rendering," in *Proc. of 9th IEEE SIES, 2014*, June 2014, pp. 160–169.

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

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

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

[12] M. Yu *et al.*, "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming," in *Proc. of the 22nd HPDC*. NY, USA: ACM, 2013.

[13] C. Basaran and K.-D. Kang, "Supporting Preemptive Task Executions and Memory Copies in GPGPUs," in *24th ECRTS*, July 2012, pp. 287–296.

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

[15] H. Janker, *Straßenverkehrsrecht: StVG, StVO, StVZO, Fahrzeug-ZulassungsVO, Fahrerlaubnis-VO, Verkehrszeichen, Bußgeldkatalog*. C.H. Beck, 2011.

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

[17] 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.

[18] S. Gansel *et al.*, "An access control concept for novel automotive HMI systems," in *Proceedings of the 19th SACMAT*, 2014.

[19] ——, "Context-aware access control in novel automotive HMI systems," in *Information Systems Security*, ser. LNCS. Springer, 2015, vol. 9478, pp. 118–138.