Institute for Parallel and Distributed Systems
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Master Thesis Nr. 3385

DEVELOPMENT OF GENERIC SCHEDULING CONCEPTS FOR OpenGL ES 2.0

Waqas Tanveer

Study Program:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Stephan Schnitzer
External Supervisor:	Dipl.-Inf. Simon Gansel (Daimler AG)
Begin date:	01.11.2012
End date:	05.07.2013
CR-Classification:	D.4.1

ABSTRACT

The ability of a Graphics Processing Unit (GPU) to do efficient and massively parallel computations makes it the choice for 3D graphic applications. It is been extensively used as a hardware accelerator to boost the performance of a single application like 3D games. However, due to increasing number of 3D rendering applications and the limiting resource constraints (especially on embedded platforms), such as cost and space, a single GPU needs to be shared between multiple concurrent applications (GPU multitasking). Especially for safety-relevant scenarios, like, e.g., automotive applications, certain Quality of Service (QoS) requirements, such as average frame rates and priorities, apply.

In this work we analyze and discuss the requirements and concepts for the scheduling of 3D rendering commands. We therefore propose our Fine-Grained Semantics Driven Scheduling (FG-SDS) concept. Since existing GPUs cannot be preempted, the execution of GPU command blocks is selectively delayed depending on the applications priorities and frame rate requirements. As FG-SDS supports and uses the OpenGL ES 2.0 rendering API it is highly portable and flexible. We have implemented FG-SGS and evaluated its performance and effectiveness on an automotive embedded system.

Our evaluations indicate that FG-SGS is able to ensure that required frame rates and deadlines of the high priority application are met, if the schedule is feasible. The overhead introduced by GPU scheduling is non-negligible but considered to be reasonable with respect to the GPU resource prioritization that we are able to achieve.

ACKNOWLEDGEMENTS

First of all I would like to thank Almighty God who has given me all the strength and the abilities to conduct this work.

I would like to pay special gratitude to my supervisor at the university, Mr. Stephan Schnitzer, who has encouraged, guided and motivated me during my thesis. He was always ready to take my questions patiently and arrange meetings whenever needed. I would also like to thank Mr. Simon Gansel for all the support he gave me at Daimler AG during this work.

Moreover, I am also thankful to Prof. Dr. Kurt Rothermel for giving me an opportunity for the thesis in his department.

I am also grateful to my family for their continuous support and encouragement which has helped me to meet my goals. Lastly, I am thankful to all those who have helped me in any way during the course of this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AFR	Average Frame Rate
FG-SDS	Fine-Grained Semantics Driven Scheduling
FRRS	Frame Rate Restricted Scheduling
GCBQ	GPU Command Block Queue
GCQ	GPU Command Queue
GPU	Graphics Processing Unit
GPU Multitasking	Execution of multiple concurrent applications on a GPU
GSQ	GPU Scheduler Queue
HPF	Highest Priority First
MTS - I	Multitasking Scenario - I
PD-FFT	Probability Distribution of Frame Finish Times
PMD	Probability of Meeting Deadlines
RT	Rendering Thread
RTC	Rendering Thread Context
ST	Scheduling Thread

1. INTRODUCTION

1.1. Motivation

As we move forward in this technological era, besides discovery of new technologies there is also a strong focus on the efficient and optimal use of those we already have. For example, in terms of computing technology on one side there is a thirst for more and more computational power on a single small chip which is being addressed in one way by scaling down individual transistor size and increasing its density [1, 2]. While on the other side, there is an effort to use this computational power efficiently for a variety of application areas such as graphics, simulations, multimedia and communications etc.

One of the parts of this latter effort is multitasking, concurrent execution of multiple applications sharing common computing resources. It is needed due to the tightness of the resource constraints, such as processing power, memory, space and cost etc., and increasing number of applications of the computing devices. The challenge in multitasking is to find a scheduling mechanism for the applications sharing computing resources, as per specified requirements such as priorities. This mechanism is typically enforced with the help of a scheduler which works on the basis of a scheduling policy. For this purpose numerous scheduling techniques [3, 4] have been developed depending on different requirements of applications and scheduling goals. This work contributes to the latter effort, as discussed earlier, by providing some scheduling concepts for the multitasking of graphic applications on a Graphics Processing Unit (GPU).

Computer graphics, like many other application areas, is growing rapidly [5] due to the rise of ubiquitous computing and the advances in processor technology. There is a wide range of graphical applications [6], such as Graphical User Interfaces (GUIs), video games, animation, advanced visualization, augmented and virtual reality etc., which facilitate enhanced user experience, entertainment and scientific research etc. Most of these applications use traditional 2D rendering techniques such as rasterization; however, due to advances in hardware capabilities 3D rendering techniques [7], such as ray tracing, bump mapping etc., are also getting pace in their usage. The applications which use 3D rendering are highly compute intensive and typically require a hardware accelerator to boost their execution. A GPU serves that purpose by working in cooperation with a Central Processing Unit (CPU).

A GPU is also a processor like CPU. However, it has more processing power due to a larger number of cores. It has the ability to perform efficient and massively parallel computations. It is extensively used as a graphics engine for 3D rendering in graphic applications such as 3D games, advanced navigation etc. It is also used for general purpose computing (GPGPU) to boost the performance of many compute intensive applications, such as advanced scientific simulations, using CUDA framework [8]. Due to these capabilities it is now an integral part, integrated or dedicated [9], of many computing systems that range from desktop PCs, laptops and handheld devices like smart phones and tablets at one end to supercomputers on the other. By having a huge computational power bank, these systems can address the computational needs of a broad scope of applications.

However, as the number of applications accelerating on GPU increase with having less flexibility in constraints such as power, space and cost especially on small handheld, mobile and embedded devices, a GPU has to allow multitasking in such a way that applications meet their Quality of Service (QoS) requirements besides maximizing GPU utilization. In case of multitasking for 3D rendering applications, QoS requirements could be average frame rates, priorities and frame deadlines (for real-time applications).

Consider an example of a future car, as shown in Figure 1.1, providing infotainment and telematics applications [10]. The displays on the dashboard are used to provide an interactive interface and seamless experience to the user/users. In this environment we could have a number of applications, such as navigation, displaying speedometer & fuel statistics, audio and/or video communication and advanced driving assistance etc. Some of these applications, such as speedometer, tachometer and navigation etc., use 3D rendering for their execution which is done with the help of GPU platform/platforms.

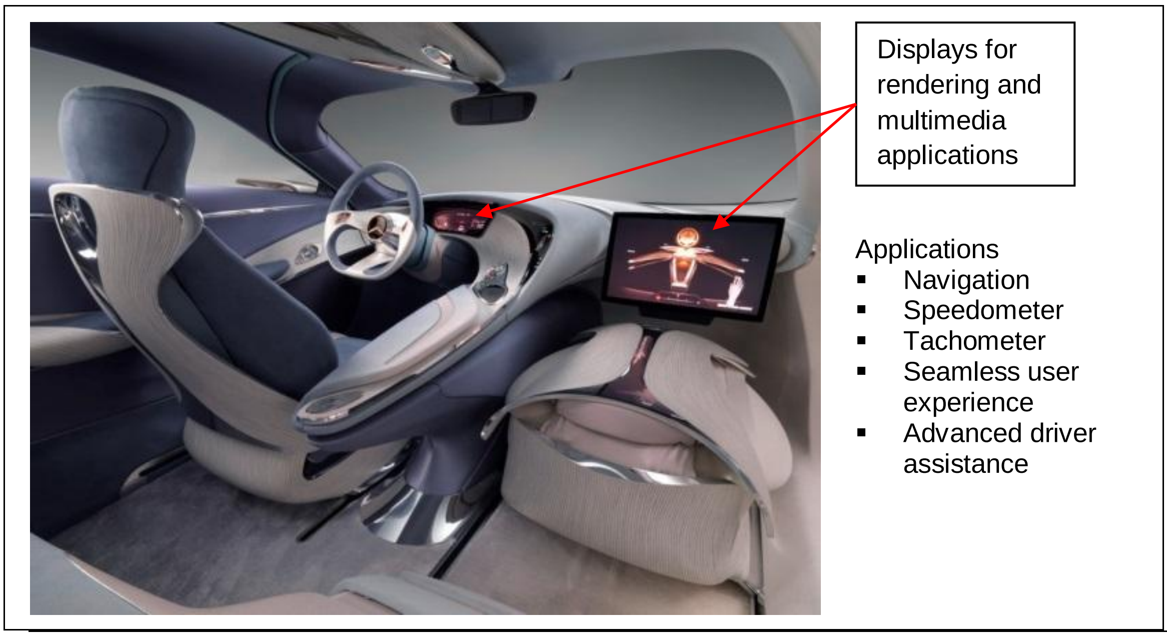

Displays for rendering and multimedia applications
Applications
▪ Navigation
▪ Speedometer
▪ Tachometer
▪ Seamless user experience
▪ Advanced driver assistance

Figure 1.1) Mercedes-Benz F125! research vehicle future telematics "@yourCOMAND" [11]

Typically, each 3D rendering application has its own QoS requirements, as discussed earlier, depending on its quality and safety. For example, the real-time applications, such as speedometer, must be prioritized over other applications such as a video call or some non real-time application. Additionally, frame rendering of these real-time applications have to be done before their deadlines, otherwise it may cause an accident due to the delayed response.

In the earlier discussion we pointed out some constraints on the computing resources, same constraints apply in this example and a single GPU has to be shared by multiple applications executing concurrently. From the user's point of view some applications may seem to be running at the same time, however, a scheduling mechanism is needed at the device end which can grant access to the applications for execution on the GPU depending on their QoS requirements.

Multitasking on a GPU can be done either in time domain [12, 13, 14] by allowing concurrent execution of multiple applications or in spatial domain [15] by partitioning GPU resources among applications and allowing them to run in parallel. This work comes under the domain of the former approach and is focused on the development of scheduling concepts for multitasking of OpenGL ES 2.0 graphic applications executing concurrently on a GPU. In order to understand the challenges of GPU scheduling we have to analyze the concepts of CPU scheduling which are not fully applicable for a GPU. These challenges are discussed in the following subsection.

1.2. Challenges in GPU Scheduling

Typically, a CPU scheduler has the flexibility of preemption, which means it can stop an application's execution at any time and grant access to another application for execution on the CPU, depending on the scheduling policy. This capability of preemptive multitasking enables a CPU to comply with the requirements of high priority applications executing concurrently with other applications. Moreover, it also enables the scheduler to grant access to an application without knowing its execution cost.

However, once a task is submitted for execution to a GPU it cannot be preempted which is an unpleasant limitation for real-time multitasking [14]. The main reason for this limitation is the dependency of GPU commands of graphic applications on specific execution context or GPU state such as transformation matrices, lighting parameters etc. Since graphic applications, such as those discussed in car example, have to maintain average frame rates and some also have real-time requirements, GPU preemption will not only result in QoS degradation but also the execution results of commands may not be correct due to their execution against incorrect or undetermined GPU state. If it occurs in time critical application then it may also result in bad consequences as well. Therefore a GPU scheduling approach is needed which has to meet QoS requirements of applications without preemption.

Additionally, the execution cost of command/commands has to be known before dispatching them to the GPU [12, 13, 14]. In this way an application can also be stopped from monopolizing the GPU resources thus allowing other applications to execute their commands on the GPU and meet their real-time requirements.

Normally a graphic application is linked to a graphics library, such as OpenGL [16] for desktop computers or OpenGL ES [17] for embedded or handheld platforms, which is part of the user-level GPU driver. Here the commands are translated to low level GPU specific commands. In Direct Rendering Infrastructure (DRI) [18], applications accumulate their commands in a command buffer before they are sent to the GPU. There are two cases in which these buffered commands are dispatched to the kernel level driver, either the command buffer is full or there is an explicit flush, using `ioctl` [13] call. The kernel level driver handles this call and enqueues the command group metadata to the ring buffer. The GPU has direct access to the ring buffer from where it takes the command group metadata and using DMA fetches the commands and executes them [13]. To the best of our knowledge, once the commands are in the ring buffer it may not be possible to change their order of execution. Therefore for GPU multi-tasking any scheduling has to be performed before the commands are inserted into the ring buffer.

There are two possibilities for the implementation of a GPU scheduler, it could be implemented either in user space or kernel space. In kernel space a better GPU utilization could be achieved as commands are enqueued in groups or batches [12, 13, 14]. Also scheduling at this coarse granularity also results in less scheduling overhead. However, because we cannot interpret command semantics at this level, it is possible that GPU gets occupied longer by an application which can delay the execution of another high priority application's commands. Also the lack of accuracy in the execution time prediction of command groups can cause errors in scheduling decisions. Therefore for real-time multi-tasking environments scheduling at this level may not be the optimal solution.

1.3. Contribution

In this thesis we have devised, implemented and evaluated a GPU scheduler in user space for multiple OpenGL ES 2.0 applications executing concurrently on a GPU.

A comparative overview of a typical system and our Fine-Grained Semantics Driven Scheduling (FG-SDS) system is presented in Figure 1.2 (a) and (b) respectively. At the top level, we have multiple OpenGL ES 2.0 applications running concurrently. For the execution of their GPU commands these applications need access to the GPU hardware, shown at the lowest level in the figure, which is provided with the help of a software mechanism consisting of a run time framework and a kernel level driver. Normally, as shown in (a), an application executes its commands by calling the native implementation of the OpenGL ES 2.0 in the user level driver. The final access to the GPU is provided by kernel level GPU driver depending on its scheduling policy. The disadvantages of scheduling at this level are already discussed in the previous section.

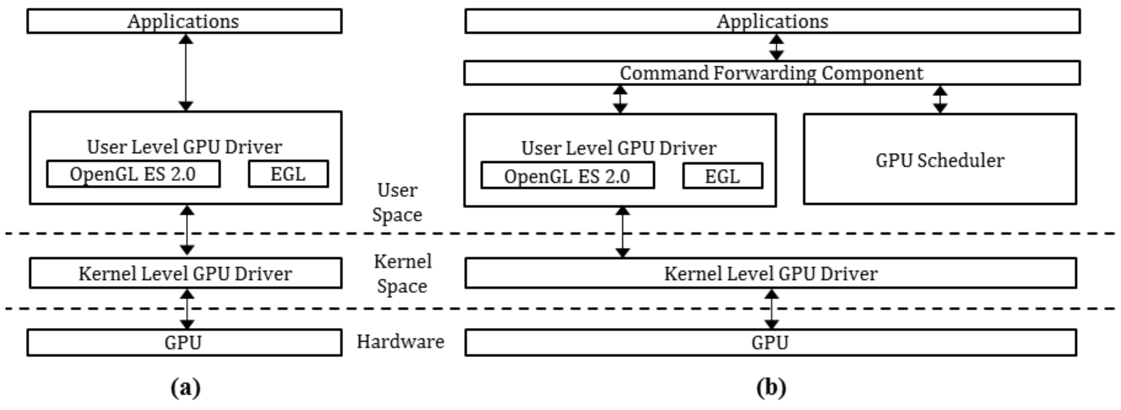


Figure 1.2) system overview (a) typical (b) Fine-Grained Semantics Driven Scheduling (FG-SDS)

In FG-SDS system, as shown in (b), instead of calling the native implementation of the OpenGL ES 2.0 directly, the commands of each application are passed to the GPU scheduler by forwarding component using shared memory. The GPU scheduler makes logical command blocks for each application's thread separately and depending on its scheduling policy selects one of these threads to dispatch their commands to the GPU. In Once an application's thread is granted permission to dispatch, it does so by calling the native implementation of the commands in its command block one by one through forwarding component. This scheduling approach is driver independent, however, it is runtime dependent and works for OpenGL ES 2.0.

The main advantage we get by scheduling in user space is the possibility to interpret individual command semantics. By using command semantics the granularity at which the scheduling is performed is fine grained to a command block level which may consist of one to a few number of commands. Moreover, it allows us to dispatch command blocks on per frame basis which enables the applications to meet their frame deadlines. It also reduces the time for which an application can occupy GPU thus allowing other applications to execute their commands. This idea uses time prediction for individual command blocks. This prediction is based on an earlier work [19] which measures execution times of different OpenGL ES 2.0 commands.

Based on these concepts, we have devised a scheduling policy known as Highest Priority First (HPF). The main features of HPF are as follows:

- Highest priority application always gets a chance to execute its commands therefore there is a high probability that it meets its frame deadline.
- Low priority applications cannot monopolize the GPU and are only allowed to execute their command blocks if the highest priority application is not ready and if its deadline is guaranteed.

- The opportunity for execution of an application follows a descending order of their priorities, therefore higher priority applications have more chance to execute than lower priority applications.
- Because of the fine-grained granularity (command blocks) of scheduling, there is a possibility for low priority applications to execute if feasible.

1.4. Organization

The rest of this report is organized in the following sequence. In Chapter 2, some related work has been presented briefly which is also focused on similar scheduling challenges of GPU multitasking. Chapter 3 discusses the concepts related to OpenGL ES 2.0, EGL, graphics pipeline and basic scheduling. The basic idea and the detailed concepts of our FG-SDS model are presented in Chapter 4. In Chapter 5, the implementation details of FG-SDS model and the HPF scheduling policy are discussed. The generic evaluation metrics and the results obtained for some multitasking scenarios are presented in Chapter 6. Finally, all the work we have done has been concluded in Chapter 7 along with its future direction.

2. RELATED WORK

We discussed in Chapter 1 that a GPU needs multi-tasking in order to cope with increasing number of applications and narrowing resource constraints. We also pointed out some of the challenges in GPU scheduling that are posed in this regard due to non-preemptive behavior of GPU execution model. We also introduced our FGS system, incorporating a GPU scheduler, to address the QoS requirements of OpenGL ES 2.0 applications. In this Chapter we briefly discuss some of the previous works, which are also focused on resolving similar issues, along with the key conceptual comparisons to our scheduling concepts. We also look in to the limitations of driver support, provided by Windows and Linux, for real-time multitasking on a GPU.

2.1. DRI

Direct Rendering Infrastructure (DRI) [18] [20] is a software framework which allows a GPU to be used directly, without making a path through the X server (although it is used to render frames to the screen), for 3D hardware acceleration on UNIX like platforms in a safe and efficient manner [18]. Primarily, it has been developed to support fast implementations using Mesa 3D graphics library [21], an open source implementation of OpenGL specification. Besides OpenGL, Mesa also provides support for OpenGL ES [21], an open source graphic library for 3D graphic applications running on embedded and handheld platforms.

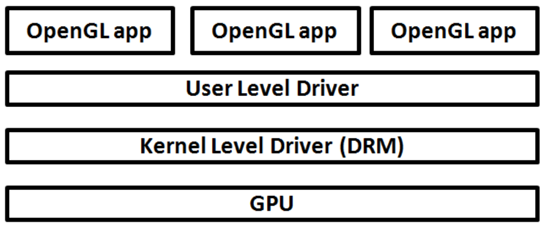


Figure 2.1) DRI - a high-level overview

Under DRI framework several open source drivers [21], such as Nouveau [22] for Nvidia graphic cards, are available. A high level overview of DRI is presented in Figure 2.1. The user level driver translates commands of a 3D graphic application to device specific commands and passes them to the kernel level driver which is also known as Direct Rendering Manager (DRM). Depending on the scheduling policy the DRM submits these commands to the command ring buffer from where they are read and executed by the GPU [13]. The next two subsections describe the modifications proposed to this basic DRI frame-work for GPU multitasking.

2.2. GERM

In order to provide a fair share, approximately same amount of GPU execution time, of GPU resources for multiple applications concurrently executing on a GPU, Graphic Engine Resource Manager (GERM) [12, 13] was introduced. It provided modifications to both user level and kernel level drivers to address the limitations of Direct Rendering Infrastructure (DRI) [20]. Prior to GERM, applications were not provided with fair share of GPU resources due to following limitations [13] of DRI:

- Before submitting commands to the kernel level driver a client application needed hardware lock to ensure isolation, one application executing its commands on a GPU against its correct graphics state. The user level driver used to fulfill this requirement for each client application. However, it made possible for a malicious application to hold lock for indefinite time thus starving other applications of their share of GPU time.
- There was also an imbalance between the times consumed by an application on a CPU and a GPU. Moreover, the granularity of scheduling was a command batch, commands sent between lock acquisition and its release by a client application. Since the execution time was not known for a command batch, could be an arbitrarily long value, some applications got more share than others.

GERM removed these limitations by introducing command queues in the kernel space for each client application. Additionally, the granularity of scheduling was also fine grained from command batch to a command group which is a group of commands sent atomically to the GPU. The GPU scheduler, with the help of command group execution time prediction, dispatched commands to the command ring buffer based on deficit round robin policy [12, 13]. It was implemented in kernel space as part of the device driver.

However, there are some key differences between GERM and FG-SDS approaches. A summary of these differences is also presented in Table 2-1. First of all, the scheduling goals of GERM are fundamentally different from those of FG-SDS. GERM’s main scheduling goal is to provide fairness between multiple applications with no real time requirements or priorities, while FG-SDS has soft real-time requirements and it also has to provide prioritization. If we also compare the granularity of scheduling in GERM, based on command groups, to that of FG-SDS, based on command blocks, the one in former is coarse-grained as compared to the latter one. Using such a coarse-grained scheduling granularity it is difficult to meet real-time requirements. Moreover, the complexity of implementation, modification to both user level and kernel level drivers, makes GERM less portable which is not the case for FG-SDS.

2.3. TimeGraph

One of the works related to our work is TimeGraph [14]. It is a scheduling mechanism implemented in the kernel space, as part of the device driver (Nouveau [22]), to support multi-tasking on a GPU in soft real-time environments. It provides prioritization and isolation for multiple OpenGL applications with the help of a GPU Command Scheduler and a GPU Reserve Manager respectively. The high priority applications are prioritized over low priority applications and at a time only one application executes on the GPU. The scheduling is performed at the granularity of GPU command groups where each such group may consist of multiple GPU commands. Moreover, for the prediction of the execution time of a GPU command group, TimeGraph also provides a GPU Command Profiler.

However, there are some key differences between TimeGraph and our FG-SDS model. The major difference is the level of implementation which is kernel space and user space in each case respectively. The other differences and benefits are intuitively derived on the basis of this basic difference. TimeGraph doesn't interpret individual command semantics whereas FG-SDS model is based on command semantics interpretation. This is one of the major benefits we get at the user space level. Due to this capability we can create fine-grained scheduling granularity of command blocks. Moreover, FG-SDS model takes care of frame deadlines in case of time critical applications whereas there is no such notion in TimeGraph.

A summary of major differences between the GERM, TimeGraph and our FG-SDS approaches are listed in the following table.

	GERM	**TimeGraph**	**FG-SDS**
1. Scheduling goals	Fairness (non real-time requirements, no priorities)	Prioritization and Isolation with real-time requirements	Prioritization and Isolation with real-time requirements
2. Implementation level	Both user space and kernel space	kernel space	user space
3. Scheduling granularity	Coarse-grained based on command groups	Coarse-grained based on command groups	Fine-grained based on command blocks
4. Frame deadlines	No notion	No notion	Yes
5. Individual command semantics	Are not interpreted	Are not interpreted	Based on the interpretation of command semantics

Table 2-1) A summanry of key differences between GERM,TimeGraph and FG-SDS

2.4. WDDM

Windows Display Driver Model (WDDM) provides basic architectural design principles for the display driver development for Windows Vista and later [23]. Typically, the graphic applications for Windows are developed using Direct3D API. However, OpenGL applications are also portable to Windows. Applications need one of these API's to communicate to the user level driver which is a Dynamic-link library [23]. The GPU access for an application is finally provided by kernel level driver, known as display miniport driver, through DirectX graphics subsystem which manages display port, video memory and GPU scheduling [23].

The commands of an application are passed to the GPU in the form of batches, which are coarse-grained with indefinite execution time, similar to the command batch idea discussed in section 2.2. If a command batch takes longer than the time it is permitted on a GPU then the GPU scheduler tries to preempt currently executing application's task by initiating a "wait" on a timeout [24]. If this task is not preempted within timeout then GPU scheduler initiates a Timeout Detection and Recovery (TDR) process to reset the GPU [24]. GPU preemption is possible at various granularities for graphics and compute applications sharing a GPU [24]. In order to avoid the initiation of TDR process the preemption granularity has be less than timeout.

To the best of our knowledge, although the preemption mechanism, as discussed above, allows high priority GPU tasks to be more responsive, however, it is not suitable for real-time applications because of following reasons [24]:

- Time taken by preemption is not known and could be indefinite; therefore there is no guarantee that high priority or time-critical applications would meet their deadlines. In worst case, GPU would be reset resulting in a system failure.
- GPU state has to be saved which adds more latency, therefore making it more difficult to meet real-time requirements.

2.5. Real-time scheduling algorithms

There are many classical scheduling algorithms [4], such as Rate-Monotonic scheduling (RMS), Earliest Deadline First (EDF) etc., which are used for real-time multitasking on a CPU. However, owing to the limitations of 3D graphical applications, such as dependency of GPU commands on specific graphic context, indefinite execution time of tasks executed on a GPU and non-preemptive nature of GPU scheduling, most of these algorithms are not applicable, as it is, in case of a GPU. In order to overcome these limitations, a new preemption mechanism is needed with a fine-grained preemption granularity. Moreover, execution time prediction of each individual grain, block of commands, is also required which can be used before the submission of a task to the GPU to stop an application from monopolizing a GPU. However, basic concepts remain the same such as deadlines, priorities etc.

3. TECHNICAL BACKGROUND

In the previous chapter, we discussed some of the related work and also presented some key conceptual differences to our work. Before going in to the discussion of our main idea, in this chapter we briefly look into some background concepts that are needed to facilitate the understanding of details presented in the next chapters.

This chapter has been divided into two major parts. The first part includes an introduction to OpenGL ES [17]/OpenGL ES 2.0 [25] and a brief description of OpenGL ES 2.0 graphics pipeline (individual stages and related concepts with some examples). It also includes a brief introduction to EGL [26], a native platform graphics interface [27]. In the second part, some basic concepts related to scheduling are discussed.

3.1. OpenGL ES and OpenGL ES 2.0

OpenGL ES [17] is an open source and cross-platform API for advanced 2D/3D graphics on embedded and handheld platforms such as smart phones, gaming consoles and others used in vehicles, avionics etc [28]. Besides providing high performance graphics, it is specifically designed to address the resource constraints, such as processing power, memory and power budget, of these platforms.

It has been derived from OpenGL [16] which is also an open source and cross-platform API for 2D/3D desktop graphics. To address the constraints, as mentioned previously, unnecessary redundancies have been removed from OpenGL API and required extensions have been added. Each version of its specification has been derived from the corresponding version of OpenGL, e.g. OpenGL ES 2.0 has been derived from OpenGL 2.0.

As we are focused on the development of scheduling concepts for OpenGL ES 2.0 specification, which consists of OpenGL ES 2.0 API specification [25] and OpenGL ES Shading Language specification (OpenGL ES SL) [29], therefore some important concepts from these specifications along with some details from OpenGL ES 2.0 Programming Guide [28] are presented in this chapter unless specified otherwise.

3.2. EGL

Typically, the graphic applications need surfaces such as windows to which they can render, which are managed by native windowing system such as X window system for UNIX like systems. EGL [26] is an interface between client graphic APIs such as OpenGL ES 2.0 and a native platform window system. Some of the major services provided by EGL to client APIs are summarized as follows [27, 28]:

- Communication mechanisms between different client APIs (e.g. OpenGL ES 2.0 and OpenVG) and the native window system.

- Management of on-screen and off-screen rendering surfaces (windows, pbuffers and pixmaps) which include their creation and sharing between different APIs. OpenGL ES 2.0 supports double-buffered windows only.
- Creation and management of graphics contexts for client applications
- Synchronization between client APIs and native windowing system

3.3. OpenGL ES 2.0 Graphics Pipeline

Typically in graphics, objects in 3D space are represented by polygon meshes which are collections of vertices, edges and faces [30]. The graphics pipeline, typically same as shown in Figure 3.1, takes these vertices with appropriate attributes, discussed in the next section, along with some other data as input and perform geometric operations specified by shaders, vertex and fragment shaders (programs), with some additional operations. The end result is a 2D image, stored in framebuffer as a pixel collection, which can be displayed on a screen such a monitor.

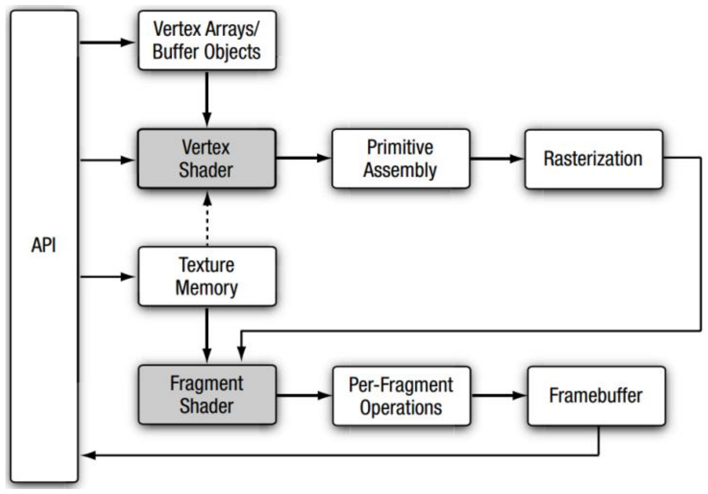


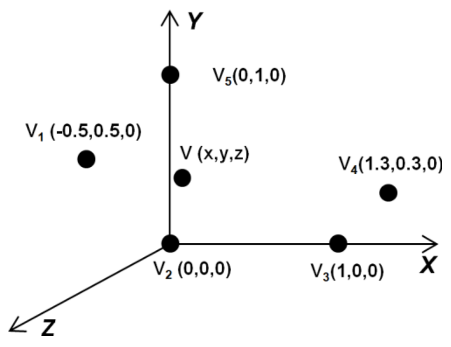
Figure 3.1) OpenGL ES 2.0 graphics pipeline [28]

The graphics pipeline of OpenGL ES 2.0 is shown in Figure 3.1. There are a number of stages in this pipeline, with grey boxes showing programmable stages. A brief description of each stage with some examples is presented in the upcoming sections.

3.3.1. Vertex Arrays and Buffer Objects

In the previous section, we discussed that vertices are the basic input to a graphics pipeline besides some other input parameters such as constants. Before any operations are performed on these vertices, they need to be stored in an efficient manner. In OpenGL ES 2.0, vertex arrays and buffer objects provide methods for storing and caching vertex data, also known as vertex attributes, respectively [28].

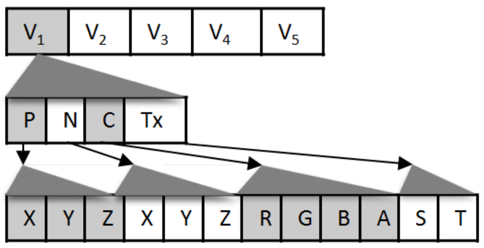
Vertex arrays are buffers mapped to the application’s address space, also known as client’s space. They are used to store per-vertex attributes in an efficient manner and are specified using `glVertexAttribPointer` function. Figure 3.2 (a) shows an example of six vertices represented in 3D coordinate space. Each of these vertices can represented by its attributes, such as position, normal, color and texture coordinates etc., as shown in Figure 3.2 (b).



Vertices (V & V1 – V5) in 3D

Vertex Attributes	
Position (P)	(x,y,z)
Normal (N)	(x,y, z)
Color (C)	(r,g,b,a)
Texture (Tx)	(s,t)

Per-vertex attributes



A way of storing vertex attributes in memory

Figure 3.2) An example of representing and storing vertex attributes

Since OpenGL ES 2.0 implements a programmable pipeline, the names of these attributes can also be specified by the application. In OpenGL ES 2.0, the minimum number of per-vertex attributes is *eight* and the maximum number is specified by `GL_MAX_VERTEX_ATTRIBS` parameter [28]. Moreover, each attribute is specified by a generic attribute index ranging from 0 to `GL_MAX_VERTEX_ATTRIBS-1`. There are two commonly used methods for specifying, allocating and storing vertex attributes, each having different performance benefits [28], which are:

- *array of structures* - all the attributes of a vertex are stored in a single buffer, contiguous memory. This method has been elaborated in Figure 3.2 (c) where v1 attributes, shown in Figure 3.2 (b), are stored in a single buffer.

- *structure of arrays* - each attribute is stored in a different buffer, not shown here.

In OpenGL ES SL, vertex attributes variables are declared in vertex shaders using `attribute` qualifier. Before a primitive, discussed in section 3.3.2, can be drawn, these attribute variables need binding to appropriate generic vertex attribute indices. This binding can either be specified by an application or by the OpenGL ES 2.0 itself. In this way vertex data is read from correct locations into the vertex shaders.

However, sometimes vertex data from vertex arrays have to be fetched into the high performance graphics memory before drawing any primitives. Vertex buffer objects are locations in graphics memory specified for this purpose. They are allocated to store or cache the vertex data. Since graphics memory offers a higher bandwidth as compared to the client's memory, buffer objects can significantly improve the performance when the same vertex data is used frequently by the draw commands. Besides vertex data, element indices specifying how the primitives are to be drawn can also be cached in to appropriate buffer objects. OpenGL ES 2.0 provides *array buffer objects* and *element array buffer objects* for storing/caching vertex data and primitive indices respectively. A generic overview of vertex arrays and buffer objects has been presented in Figure 3.3.

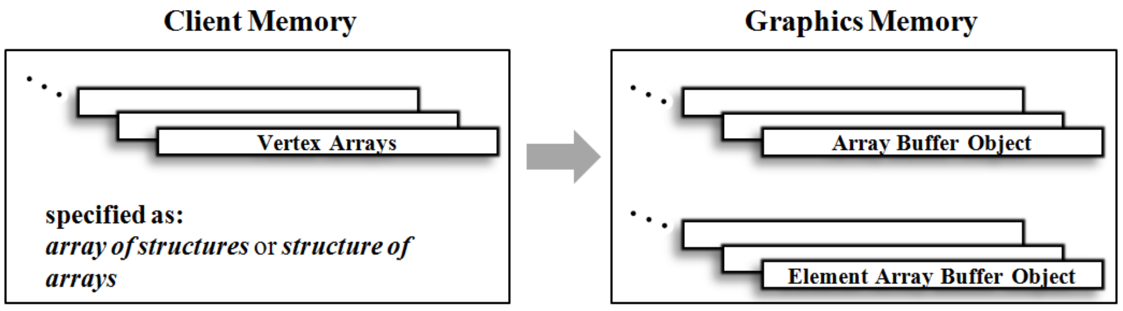

Client Memory
Vertex Arrays
specified as:
array of structures or *structure of arrays*
Graphics Memory
Array Buffer Object
Element Array Buffer Object

Figure 3.3) An overview of data objects in client and graphics memory

The next stage in the graphics pipeline is vertex shader (cf. section 0), however, primitive assembly and rasterization stages are discussed next due to the fact that the type of individual primitives that are to be drawn and their corresponding vertex indices are specified before they are actually passed to the vertex shader.

3.3.2. Primitive Assembly

In this stage of the graphics pipeline various primitives, basic geometric objects, are drawn by assembling vertices, transformed by vertex shader, in a specified order. Prior to the explanation of operations that are performed in this stage, these primitives are discussed below.

3.3.2.1. Primitives

In OpenGL ES 2.0 three kinds of primitives are supported which are triangles, lines and points sprites. These primitives can be drawn by executing `glDrawArrays` or `glDrawElements` command. A brief description of each primitive type along with its subtypes, if any, is presented in the next three subsections.

3.3.2.1.1. Triangles

There are three ways in OpenGL ES 2.0 by which triangles primitives can be drawn. An example for each one, named by its specific type, is shown in Figure 3.4 (a), (b) and (c). In Figure 3.4 (a) two triangles are drawn using two sets, (V_1, V_2, V_3) and (V_4, V_5, V_6), of vertices with each one consisting of three unique and adjacent vertices. This mode is specified as `GL_TRIANGLES`.

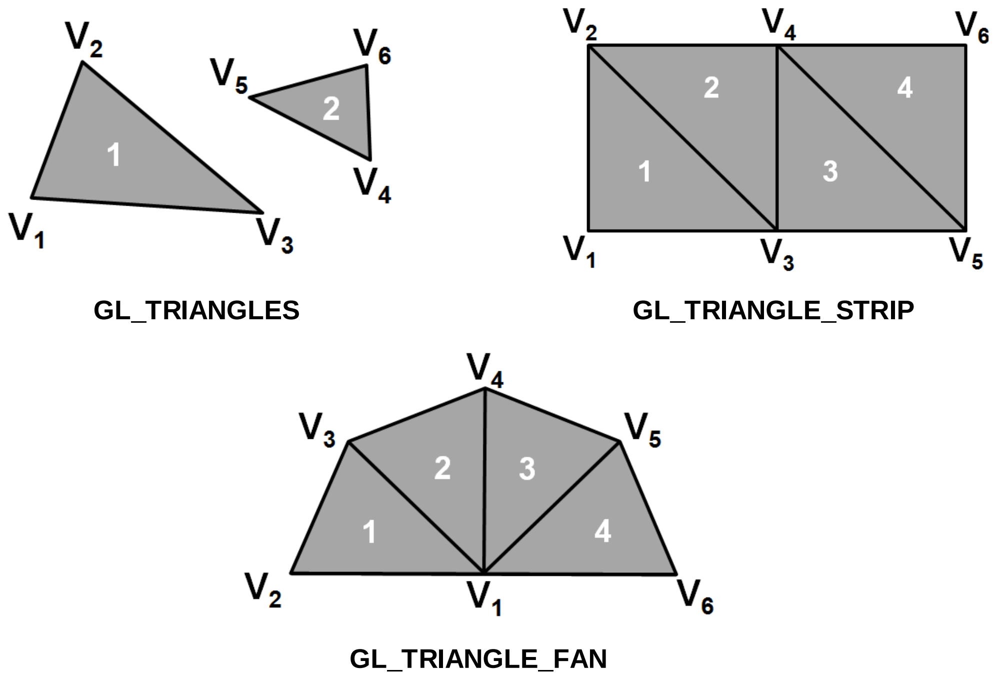


Figure 3.4) Examples of triangle primitive types in OpenGL ES 2.0

Similarly in (b), four triangles are drawn with (V_1, V_2, V_3), (V_2, V_3, V_4), (V_3, V_4, V_5) and (V_1, V_2, V_3) sets of vertices respectively, using primitive mode `GL_TRIANGLE_STRIP`. Note that in each case of two consecutive triangles, two vertices are common between the two sets, the last two from the first one and the first two from the second one. Another type of triangle primitive is specified as `GL_TRIANGLE_STRIP` which is shown in Figure 3.4 (c). In this case four triangles are drawn by (V_1, V_2, V_3), (V_1, V_3, V_4), (V_1, V_4, V_5) and (V_1, V_5, V_6) sets of vertices respectively. As its name suggests, a fan is drawn with the common vertex between all the sets at the center and all other vertices making its wings. Note the relation between two adjacent sets of vertices in each case.

3.3.2.1.2. Lines

Figure 3.5, (a), (b) and (c), shows three ways in which line primitives can be drawn in OpenGL ES 2.0. In the first case, i-e (a) specified by `GL_LINES`, three separate line segments are drawn with (V_1, V_2), (V_3, V_4) and (V_5, V_6) sets of vertices. The next one, shown in (b), is specified as `GL_LINE_STRIP`. In this case five connected lines are drawn by connecting each vertex with its preceding vertex, with the exception of the vertex with a starting index.

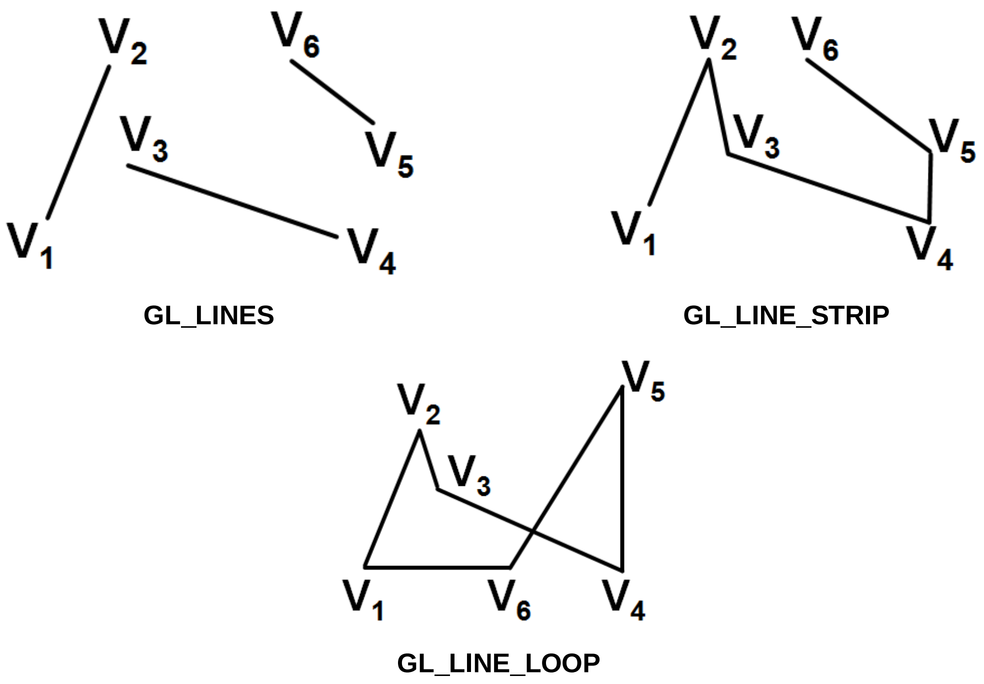


Figure 3.5) Examples of line primitive types in OpenGL ES 2.0

The third case, shown in (c) is similar to (b), however, the vertex with the last index (6) is also connected to the vertex with the starting index (1). It is specified as `GL_LINE_LOOP`.

3.3.2.1.3. Point sprites

Each vertex can also be drawn separately with the help of `GL_POINTS` primitive type. An example of this type is shown in Figure 3.6 where 20 vertices are drawn.

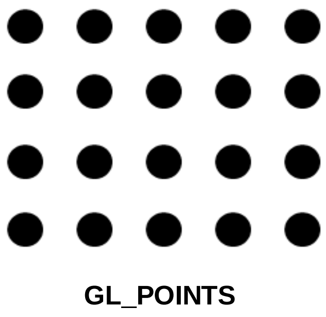


Figure 3.6) Example of points primitive type in OpenGL ES 2.0

3.3.2.2. Primitive Assembly Operations

As the vertices are processed through different stages of the graphics pipeline, their coordinates such as position coordinates undergo various transformations. Figure 3.7 gives an overview of the transformations which are performed as the vertices are processed through the vertex shader and the primitive assembly stages. Moreover, it also gives information about the operations that are performed during primitive assembly stage.

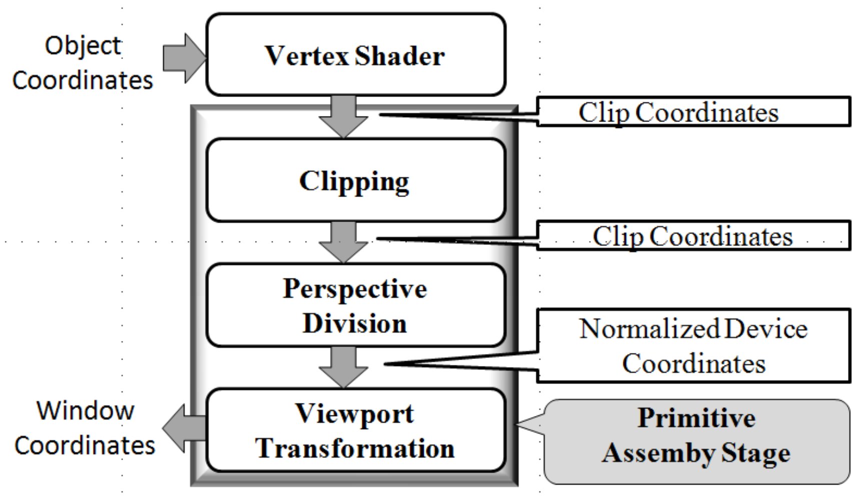


Figure 3.7) Primitive assembly stage operations and corresponding coordinate systems

In the real world objects are typically represented or modeled in object or local coordinate system as shown in Figure 3.2 (a). Referring to Figure 3.7, it can be seen that the input vertices to the vertex shader are in this coordinate system. After the processing by the vertex shader the vertices are in clip coordinates and are passed to the primitive assembly stage which comprises clipping, perspective division and the viewport transformation operations.

In the following subsections these operations and their corresponding coordinate systems are described briefly. In each case the example is given from the vertex position attribute represented as $p(x, y, z, w)$.

3.3.2.2.1. Clipping

The input and output vertex coordinates in this operation remain in the same coordinate system which is clip coordinate system as shown in Figure 3.7. However, considering position of a vertex which is represented as $p(x_c, y_c, z_c, w_c)$ in this coordinate system, each primitive, such as triangle etc., is clipped to a viewing volume as shown in Figure 3.8.

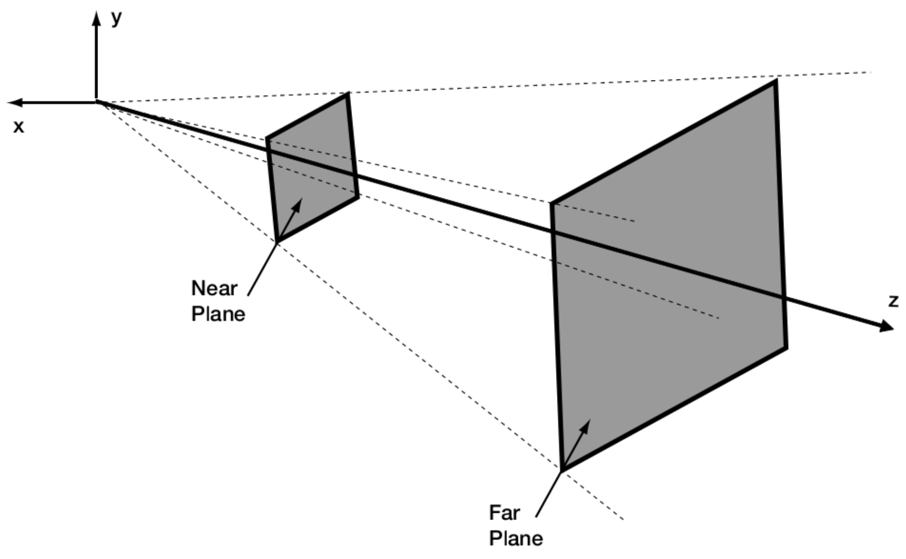


Figure 3.8) Viewing volume for clipping [28]

The viewing volume is bounded by six clipping planes which are briefly described as below:

- In the direction of line of sight (along z-axis), a near plane and a far plane as shown in Figure 3.8.
- A top and a bottom plane along y-axis.
- A left and a right plane along x-axis.

The clipping planes discussed above can be found out with the help of following equations which set bounds on clip coordinates.

$$-w_c \leq x_c \leq w_c \qquad \cdots\cdots \quad \textit{Equation 3-1 } [28]$$

$$-w_c \leq y_c \leq w_c \qquad \cdots\cdots \quad \textit{Equation 3-2 } [28]$$

$$-w_c \leq z_c \leq w_c \qquad \cdots\cdots \quad \textit{Equation 3-3 } [28]$$

Once the clipping planes are known each individual primitive goes through clipping operation separately and the portion which is outside the viewing volume is clipped away. In some cases, such as for triangles or lines, clipping could result in the generation of new vertices as well. However, if a primitive is completely inside the clipping volume it passes as it is to the output. Similarly, if it is completely outside it is discarded away.

3.3.2.2.2. Perspective Division

The next operation after clipping in the primitive assembly stage is the perspective division as shown in Figure 3.7. With this operation vertex coordinates from clip coordinate system are transformed to the normalized device space. The transformation through perspective division for the position coordinates (x_c, y_c, z_c, w_c) is depicted in Figure 3.9.

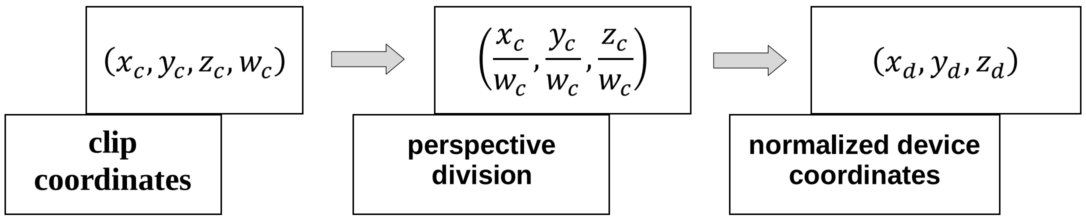

(x_c, y_c, z_c, w_c)
clip coordinates
$\left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c}\right)$
perspective division
(x_d, y_d, z_d)
normalized device coordinates

Figure 3.9) A depiction of perspective division [28]

The normalized device coordinate system has a range specified as from $[-1 \ ... + 1]$.

3.3.2.2.3. Viewport Transformation

The viewport transformation is the last operation of the primitive assembly stage, as shown in Figure 3.7. It transforms vertex coordinates from normalized device coordinate system to the window coordinate system of the window to which an application renders. Figure 3.10 specifies the mathematical operation on each vertex coordinate as it goes through this operation.

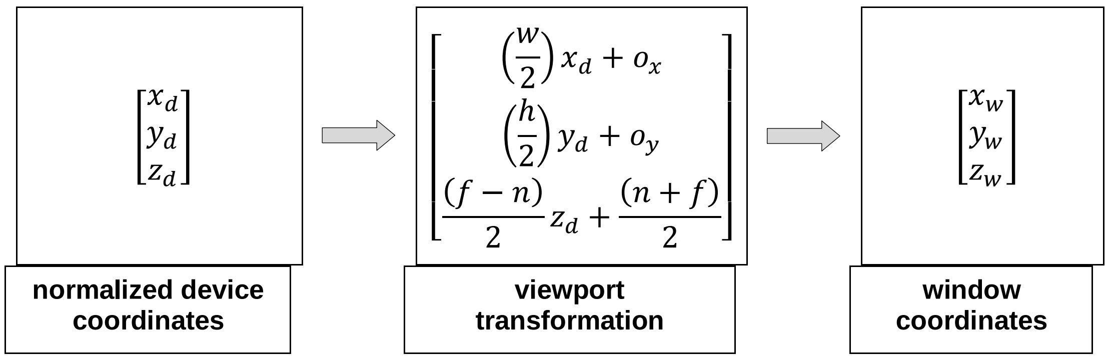

$\begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix}$
normalized device coordinates
$\begin{bmatrix} \left(\frac{w}{2}\right) x_d + o_x \\ \left(\frac{h}{2}\right) y_d + o_y \\ \frac{(f-n)}{2} z_d + \frac{(n+f)}{2} \end{bmatrix}$
viewport transformation
$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix}$
window coordinates

Figure 3.10) viewport transformation operation [28]

In the central part of the figure,

- w and h represent the width and height of a window respectively, an example is shown in Figure 3.11.
- $o_x = \frac{(x+w)}{2}, o_y = \frac{(y+h)}{2}$
- n and f represent the depth range values with a specified range of $(0.0, 1.0)$. The values outside this range are clamped to the boundary values which are also the default value.

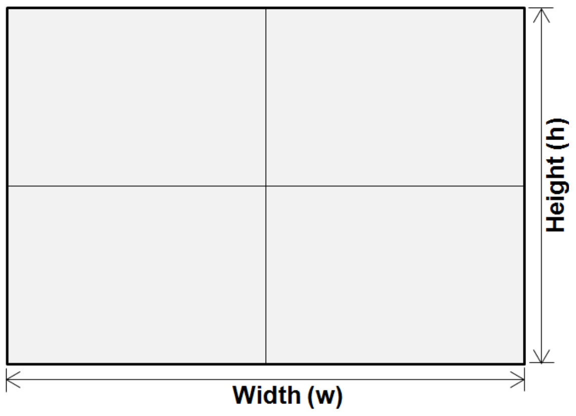
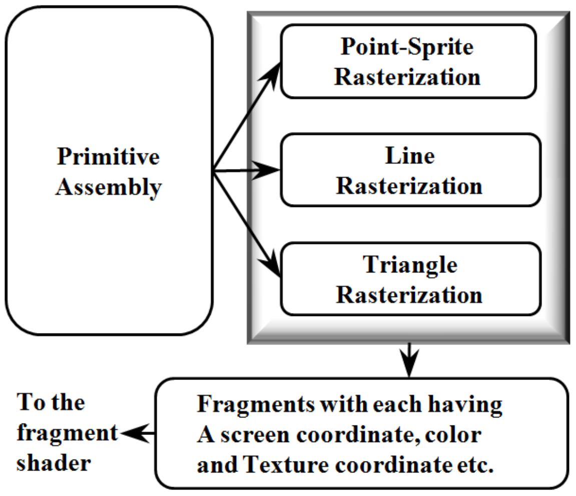


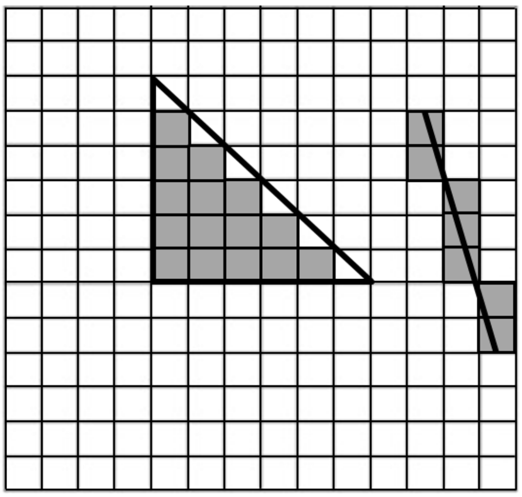
Figure 3.11) A depiction of a window

3.3.3. Rasterization

This stage of the graphics pipeline, as shown in Figure 3.12 (a), transforms individual primitives, such as triangles, lines etc., to a 2D image of fragments. Each such fragment has a unique location represented in window coordinates as (x_w, y_w). However, unlike the coordinates, represented by floating point values which come out from the primitive assembly stage, these coordinates are represented by integral values. In addition to the position coordinates, each fragment holds some other data values which is used by the fragment shader for further per-fragment procressing.


Primitive Assembly
Point-Sprite Rasterization
Line Rasterization
Triangle Rasterization
Fragments with each having A screen coordinate, color and Texture coordinate etc.
To the fragment shader

Rasterization stage [28]



An example of primitive rasterization

Figure 3.12) An overview of rasterization stage with an example

The process of rasterization is also illustrated with the help of an example as shown in Figure 3.12 (b). In this example, a triangle and a line primitive are rasterized which is represented by the shaded squares of the grid in each case.

In addition to fragment generation, for primitives like triangles rasterization stage also determines whether a primitive be rasterized at all or not. For this purpose each primitive goes through culling process which is discussed below.

3.3.3.1.1. Culling

In order to avoid unnecessary rasterization, culling process helps determining whether a triangle primitive is front-facing or back-facing. The primitives which are visible to the viewer are called front-facing and the primitives that are hidden are known as back-facing. Once it is known, rasterization is only performed for the front-facing primitives.

For this purpose, orientation of the triangle has to be known which is determined by calculating the signed area of the primitive, under culling process, in the window coordinates. The orientation of a triangle primitive could either be clockwise or counter-clockwise and which orientation represents front-facing or back-facing primitives can be specified by the application. Thus by using appropriate commands the rasterization for the specified primitives, that are to be culled, could be avoided.

Before the discussion of a fragment shader, we first discuss the operations performed by a vertex shader.

3.3.4. Vertex Shader

A vertex shader, as shown in Figure 3.13, is a graphics pipeline stage which implements a general programmable function. The input to this function includes parameters such as per-vertex attributes, uniforms and samplers. Typically, this function applies various mathematical operations on the input data such as position transformations, computation of lighting equation to generate per-vertex color values etc. The output of this stage is 1:1 mapped with respect to the input and the output variables are known as varyings.

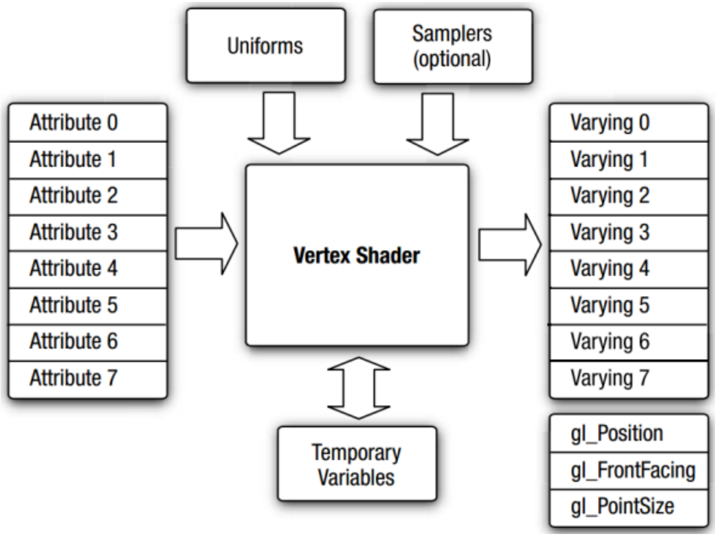
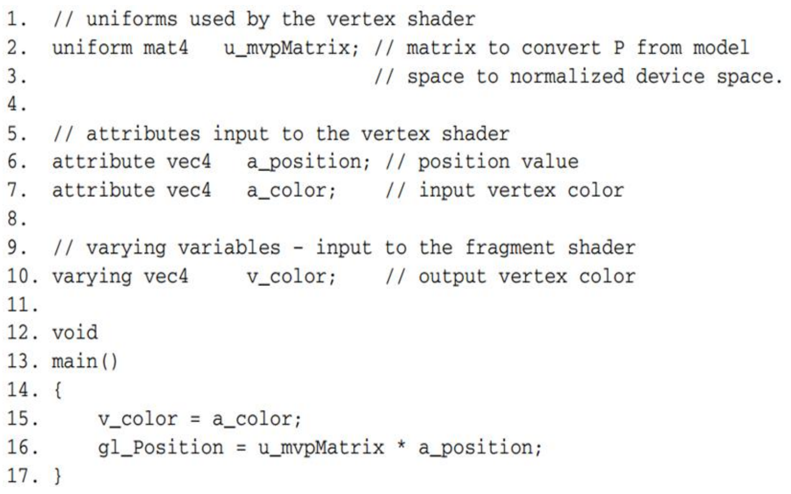


Figure 3.13) OpenGL ES 2.0 vertex shader, overview [28]



```
1.  // uniforms used by the vertex shader
2.  uniform mat4   u_mvpMatrix; // matrix to convert P from model
3.                              // space to normalized device space.
4.
5.  // attributes input to the vertex shader
6.  attribute vec4   a_position; // position value
7.  attribute vec4   a_color;    // input vertex color
8.
9.  // varying variables - input to the fragment shader
10. varying vec4     v_color;    // output vertex color
11.
12. void
13. main()
14. {
15.     v_color = a_color;
16.     gl_Position = u_mvpMatrix * a_position;
17. }
```

Figure 3.14) OpenGL ES 2.0 vertex shader, example [28]

A simple example of a vertex shader, written in OpenGL ES SL, is shown in Figure 3.14. The input parameters, shown in Figure 3.13 and also specified from line 1 to line 7 in Figure 3.14, to a vertex shader are briefly discussed as follows:

Attributes: Each vertex is represented by its attributes, as discussed in the previous section and also shown in Figure 3.2 (b). The input attributes to the vertex shader shown in Figure 3.13 are represented by generic attribute indices from 0 - 7. In Figure 3.14, line 6 and 7 indicate position and color attributes of a vertex.

Uniforms: These are the constants that are used by the vertex shader. Typically, a vertex shader can take the vertex attribute values either from vertex arrays or from these uniforms. In Figure 3.14 line 2 indicates a uniform, which is actually a transformation matrix.

Samplers: These are special type of uniforms which represent textures.

Shader program: It is an computer program that is executed on each vertex.

The shader program applies the required operations and writes output varying variables accordingly. In Figure 3.13, output of the vertex shader is specified by varying variables denoted by indices 0 – 7. In Figure 3.14 , line 10 shows a single varying variable which represents color information.

In addition to the input and output variables discussed earlier, there are also some built-in variables, such as `gl_Position` shown in Figure 3.13 and also specified in line 16 of Figure 3.14, used to store some specific information. Moreover, temporary variables may also be declared and used by a vertex shader.

3.4. Basic Scheduling Concepts

The idea of scheduling is not so uncommon and is being used in our daily lives. It's all about how well can we manage the time and resources we have to do our tasks. A fair schedule helps us to do more tasks and makes us more productive. If we think for a while, we see that all the tasks we have to do don't have same priorities. Some tasks are more urgent than others and have to be done in time; otherwise we could miss their deadlines which in some cases could even lead to bad consequences. Therefore, these tasks need our special attention so that we don't miss their deadlines while being busy with some unimportant task. Even though some tasks might not be that much important, if we just forget doing them makes us less productive. In order to avoid forgetting some important tasks, like an appointment for example, we insert calendar entries by mentioning its time thus making sure its execution in future. That's how we use these scheduling concepts.

The use of scheduling in the context of GPU resource management is also not so different. As we have discussed in Chapter 1, multiple concurrent applications may not leverage GPU's high computational power efficiently without having a scheduler. A GPU scheduler does that job by providing a fair share of GPU resources to applications as per specifications. The specifications of the applications, such as frame rates and priorities, influence the scheduling policy used by the scheduling algorithm. In order to address QoS (Quality of service) requirements of applications these specifications must be met. In this chapter, we discuss in detail some generic scheduling concepts and those we have devised for OpenGL ES 2.0 applications.

3.4.1. A Generic Scheduling Scenario

A generic scheduling scenario with some usual requirements and goals is depicted in Figure 3.15. We have *N* number of tasks which are to be executed on a system with *M* number of resources, as shown in Figure 3.15 (a) and (c) respectively. *M* is usually less than *N* and also a task may use 1 to *M* resources at a time for its execution. It constrains the order of execution of these tasks and limits the amount of time a task can grab some resource or resources. The scheduler, shown Figure 3.15 (b), implements a scheduling algorithm which takes some scheduling parameters of these tasks such as their periods, execution times etc. as input and finds a solution that can meet some scheduling goals, as shown in Figure 3.15 (d).

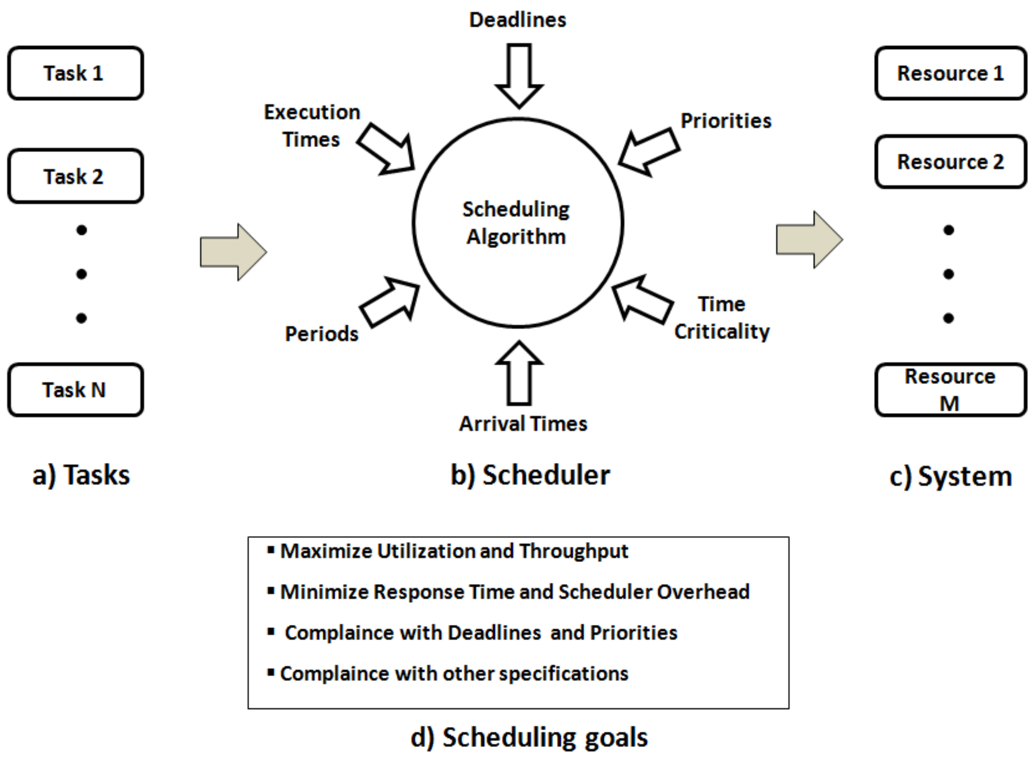


Figure 3.15) Generic scheduling scenario with some requirements and goals

A general scheduling goal is to maximize the utilization of the system which means that the system must be doing some useful work most of the time. Another goal is to maximize throughput, number of tasks done per unit time, which helps a system to execute more than one task concurrently. It is also desired to minimize scheduling overhead and response time of the system, which result in greater utilization and makes the system lively respectively. Some tasks may be more important than others which is specified by giving priorities to individual tasks. For real time tasks, it is also required to meet deadlines. In hard real-time systems missing deadlines could result in system failure or even loss of life. However, in soft real-time systems it could result in some quality degradation. In general, scheduling algorithm has to find a fair tradeoff between these scheduling goals.

4. FINE-GRAINED SEMANTICS DRIVEN SCHEDULING

So far we have discussed the importance and the challenges of GPU multitasking. We also briefly introduced our scheduling approach. Some basic concepts regarding OpenGL ES 2.0 and scheduling are also described.

In this chapter, we discuss our Fine-Grained Semantics Driven Scheduling (FG-SDS) model in detail. The organization of this chapter is as follows. In the first subsection the core idea of our work has been presented. In the next subsection, the complete system model is described which illustrates the way in which the scheduler is integrated within the system and how the components interact with each other. Finally a sample GPU multitasking scenario has been chosen to describe the way in which our idea works.

In the discussion that follows in this section when we refer to the applications, it is implicit that these are OpenGL ES 2.0 applications.

4.1. Basic Idea

In order to present the basic idea of our work, we proceed as follows. Firstly, the scheduling goals of the FG-SDS model are discussed. Secondly, the challenges in achieving these goals are described briefly. Finally, we discuss how the FG-SDS model can overcome these challenges to meet the scheduling goals.

4.1.1. Scheduling Goals

As we discussed in section 1 that applications have some QoS requirements, considering these requirements the major scheduling goals of FG-SDS model are as follows:

Prioritization: The high priority applications have to be prioritized over the low priority applications.

Maintaining Average Frame Rates: Each application has to maintain a desired Average Frame Rate (AFR).

Meeting Frame Deadlines: Besides maintaining an average frame rate, the time critical applications also have to meet their frame deadlines.

4.1.2. Scheduling challenges

The key challenges to meet the scheduling goals, which are mentioned above, are as follows.

No Preemption: The GPU execution model is non-preemptive. Each application has to be executed with correct graphics context and during the execution it cannot be preempted. Therefore scheduling has to be performed without preemption.

Indefinite Execution Time Cost: To avoid the occupation of the GPU resources by an application for an indefinite time, the execution time cost for a task must be known before dispatching its commands to the GPU.

4.1.3. FG-SDS

As its name implies, the FG-SDS model is an approach of GPU scheduling by splitting the tasks of an application into fine grain components, also known as command blocks, using the command semantics. The core features of this model are as follows.

Fine-Grained Scheduling Granularity - The scheduling granularity has been reduced from a command group (coarse-grained), which is used in [12, 13, 14], to a command block (fine-grained) level in FG-SDS model. The number of commands in a command block can range from one command, which is the minimum possible scheduling granularity, at least to a few commands at most. Reducing the granularity to such a level results in the reduction of time for which an application can occupy a GPU whenever it is allowed to execute. After dispatching the commands of a command block to the GPU, an application safely returns back the control to the scheduler.

In this way the limitation of non-preemption, regarding the GPU execution model, has been addressed in an efficient manner. On one hand it creates more possibilities for the scheduler to address the requirements of high priority applications. On the other hand it allows the low priority applications to dispatch their commands without affecting the high priority applications.

Using the Predicted Execution Times of the Command Blocks - The FG-SDS model also has the knowledge of the predicted execution time for each command block. However, the prediction is not the part of FG-SDS. It has been done in an earlier work [19]. This availability of this knowledge in advance helps FG-SDS model to grant permission to low priority applications in such a way that the high priority applications do not miss their deadlines. It creates a kind of determinism in the system.

Using Command Semantics - The type of commands to be included in a command block, its size and the dispatch procedure is based on the interpretation of command semantics. The knowledge of command semantics helps in splitting the tasks and making command blocks in such a way that each application renders correctly during the interleaving.

4.2. System Model

A complete overview of the system model has been presented in Figure 4.1. The top most layer represent the N number of applications which need a concurrent access to the GPU for the execution of their commands. Each application may further have several rendering threads of execution.

Typically, in order to execute their commands each application's thread calls a function in the user level driver or the native implementation. The user level driver converts the function call in to the GPU commands which are dispatched to the GPU through the kernel level driver.

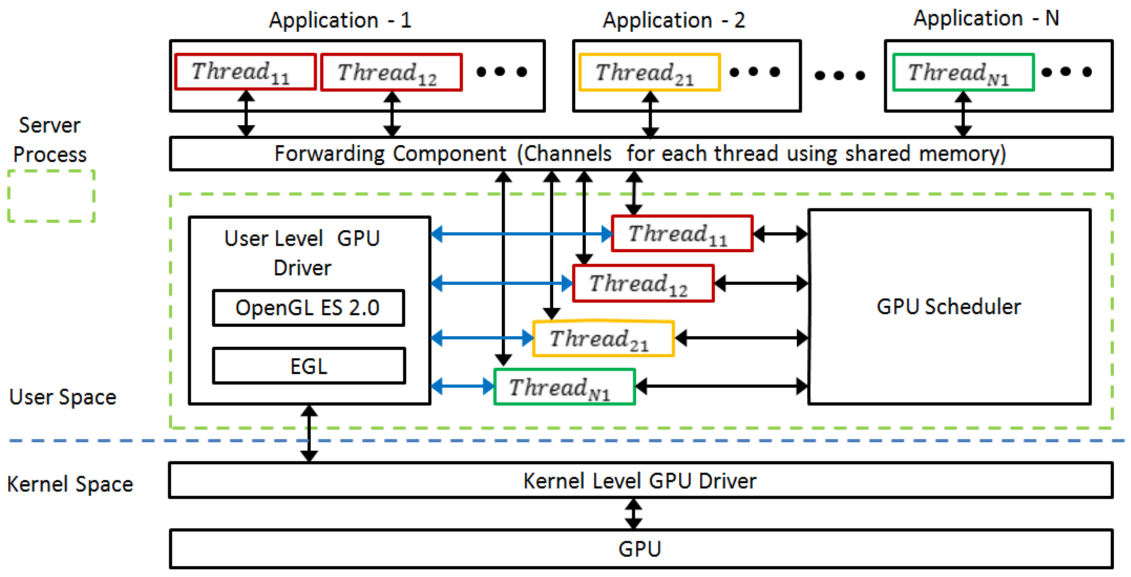


Figure 4.1) System Model

However, in our system model the function calls by the application threads are directed to the server process through a forwarding component. The forwarding component provides communication channels between the applications and the server process using shared memory. The server process creates a separate thread for each corresponding application's thread locally which can also be considered as a duplicate of it regarding their execution behaviour. Now each of these created threads of the server process communicate with its corresponding application's thread using its assigned communication channel. These threads are responsible for dispatching commands of their corresponding applications to the GPU. For this purpose they have to take permission from the GPU Scheduler. In this way applications communicate with the GPU Scheduler and forward their commands to it.

The GPU Scheduler provides a queue for each thread where it can temporarily place its commands. In addition to these queues, the GPU scheduler also provides appropriate functions for the application threads to arrange their commands in to the command blocks by using the forwarded command semantics. Similarly, an application thread becomes ready to dispatch its command blocks to the GPU after it forwards commands of some specific semantics such as the command which changes the contents of frame buffer (`eglSwapBuffers`) or a synchronous command. At this point the thread stops forwarding commands and waits for the scheduler permission to dispatch its command blocks. The GPU scheduler has a separate scheduling thread which then grants permission to an application's thread to dispatch by using a specified scheduling policy.

When an application's thread is allowed to dispatch, it calls the native implementation in the user-level drive and then it follows the typical execution behavior until the command block under consideration is executed on the GPU. However, an application cannot dispatch indefinite commands to the GPU. The number of command blocks to be dispatched at a time is decided by the scheduler and then the allowed thread can only dispatch those command blocks. The order of the commands for a single thread does not change no matter how long it waits or how often it dispatches.

4.3. A Sample GPU Multitasking Scenario

To understand the FG-SDS model, consider a sample application set consisting of three OpenGL ES 2.0 graphic applications with some QoS parameters as specified in Table 4-1. Each application has a priority and an average required frame rate. Moreover, each application has a frame period which is derived from its desired average frame rate. Let's also assume that these applications have soft real-time requirements of meeting their frame deadlines as well. The frame deadlines for an application can be derived as follows:

$$FD: Frame\ Deadline\ , FP: Frame\ Period$$

$$FD_i = FP_i * F_{ij} \quad \dots \quad eq.(1)$$

$$i = Application\ Id\ , \qquad F_{ij} = jth\ frame\ of\ ith\ application$$

Application	**Priority**	**Frame Rate (FPS)**	**Frame Period [1/FPS] (ms)**
1	High	50	20
2	Medium	25	40
3	Low	20	50

Table 4-1) sample soft real-time graphic application set with some specified QoS parameters

By using the above information we have to schedule these applications to execute on the GPU with our FG-SDS model. Figure 4.2 shows a scheduling approach for the multitasking of applications in the sample set based on FG-SDS model.

In order to simply the scenario we assume that each frame of an application is composed of three command blocks. It is also assumed that each application has to render the same frame periodically. The required execution time for each command block of an application is also known. Moreover, we also assume that command blocks are executed on the GPU immediately after they are dispatched by an application.

Figure Legend:

↑	First Frame Arrival Time	↓	Frame deadline/Next Frame Arrival Time
CB	Command Block	▯	Scheduler Overhead

	Required Execution Time			
	Frame	CB-1	CB-2	CB-3
A1: Application-1	🟥🟥🟥	🟥	🟥	🟥
A2: Application-2	🟩🟩🟩	🟩	🟩	🟩
A3: Application-3	🟦🟦🟦	🟦	🟦	🟦

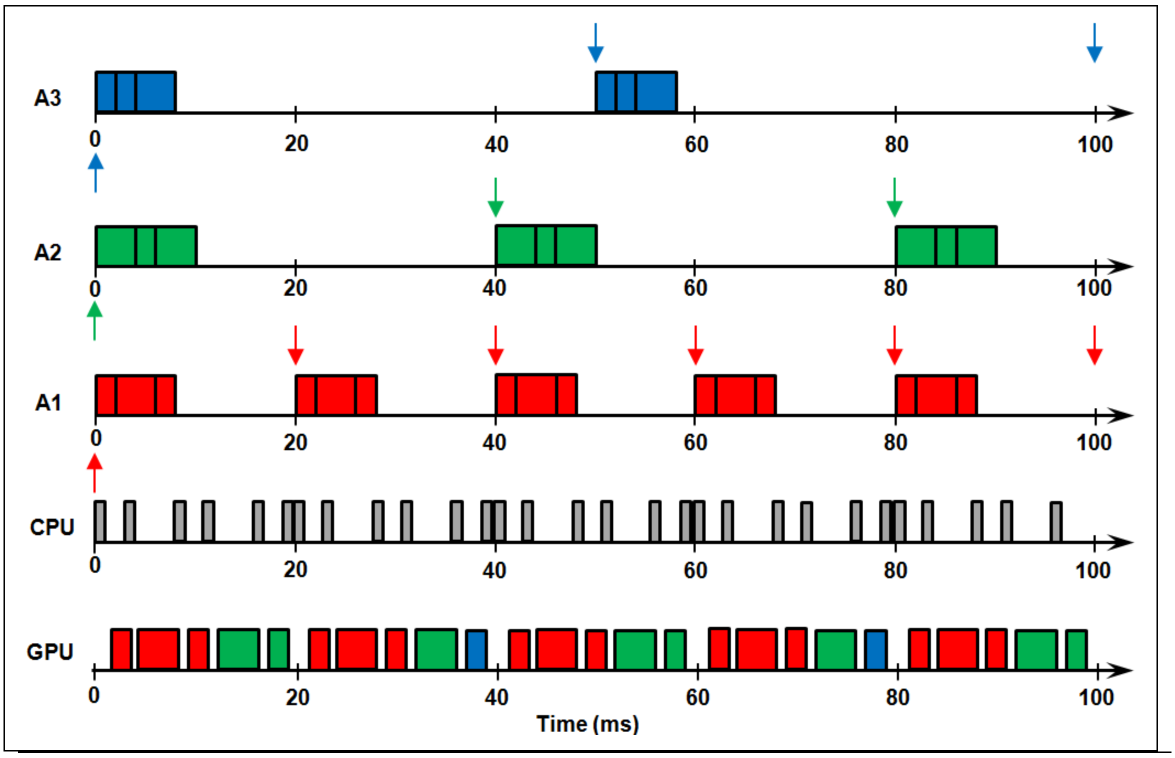


Figure 4.2) Frame rendering of a sample graphic application set using GPU

In the figure above, the command blocks of a frame along with the frame arrival times for each application, A1, A2 and A3, are shown on the timeline. The deadline for each frame is also indicated which is also the arrival time for the next frame. As we mentioned in the previous subsection that each applications has its own queue where it enqueues its command blocks, although in the above figure these queues are not shown but we assume that command blocks are inserted in to these queues before being dispatched by an application. The scheduler executing on the CPU selects one application from the sample set and grants it the permission to dispatch its command block to the GPU.

For example in the start at time 0, all the applications of the sample set are ready to dispatch their commands. However, the scheduler selects A1, which has a High priority, and allows it to dispatch. During this scheduling decision the scheduler takes some time which is known as scheduling overhead. It is shown on the CPU time-line. The A1 after dispatching its first command block sends an acknowledgement to the scheduler that it has dispatched. The scheduler selects A1 in the next two scheduling decision as well as it is the highest priority application. After A1 finishes its first frame, the next application with medium priority is A2 which is given permission the next two times in which a scheduling decision is made.

As the time reaches 20ms, A1 gets ready again to dispatch its next frame command blocks. Again it is selected by the scheduler until its current frame ends. Then A2 is selected. At this point only A3 is ready to dispatch. Now the scheduler selects this low priority application to dispatch. In this way the applications are selected and are given the permission to dispatch.

Within 100 ms, which is the end time shown on the time-line, only A1 and A2 are able to render their frames before their deadlines. While A3 only manages to dispatch the two command blocks of its first frame.

In this way FG-SDS model allows high priority applications to meet their deadlines without allowing low priority applications to occupy the GPU.

5. IMPLEMENTATION

5.1. Overview

This chapter discusses the implementation details of our FG-SDS model. The implementation architecture of this model is presented in Figure 5.1. The shaded boxes (without color) represent the active components (threads) which can communicate with each other and also perform specified operations on passive components (queues). They can write to (shown with an arrow pointing to the queue to be written) or read from (shown with an arrow pointing away from the queue to be read) the respective queues. Moreover, all the threads shown in the figure belong to the same process. For the sake of clarity, the upper layers (forwarding layer and application layer) are not shown in this diagram. The major components of this architecture are described as follows.

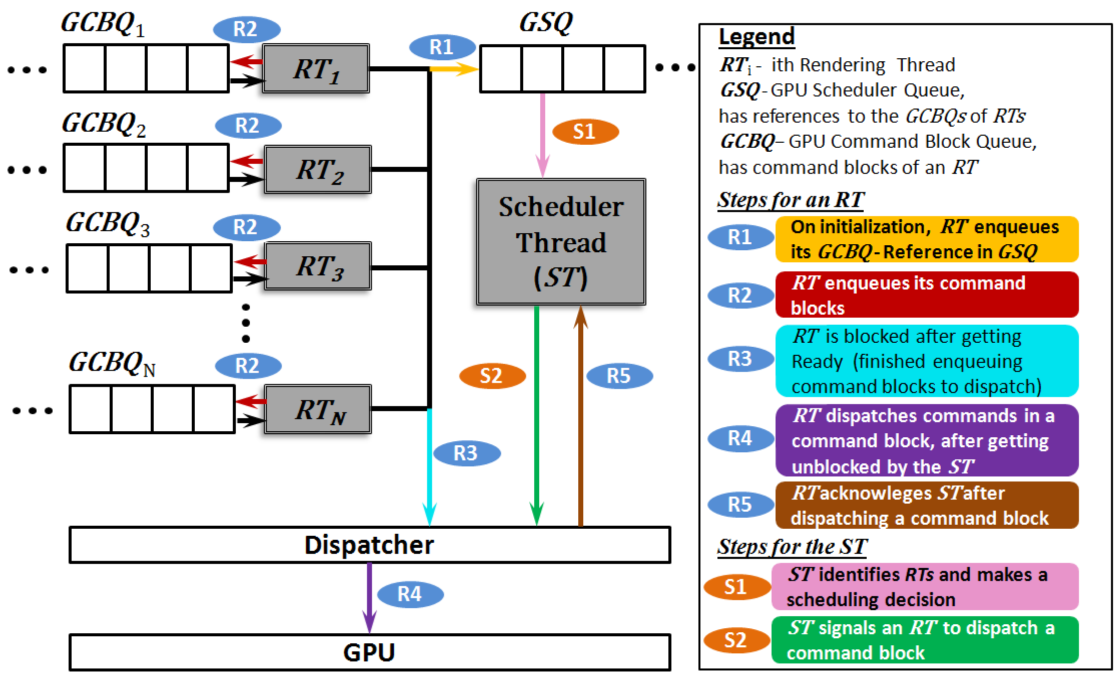

GCBQ1
R2
RT1
R1
GSQ
GCBQ2
RT2
S1
GCBQ3
RT3
Scheduler Thread (ST)
GCBQN
RTN
S2
R5
R3
Dispatcher
R4
GPU
Legend
RTi - ith Rendering Thread
GSQ - GPU Scheduler Queue, has references to the GCBQs of RTs
GCBQ – GPU Command Block Queue, has command blocks of an RT
Steps for an RT
R1 On initialization, RT enqueues its GCBQ-Reference in GSQ
R2 RT enqueues its command blocks
R3 RT is blocked after getting Ready (finished enqueuing command blocks to dispatch)
R4 RT dispatches commands in a command block, after getting unblocked by the ST
R5 RT acknowleges ST after dispatching a command block
Steps for the ST
S1 ST identifies RTs and makes a scheduling decision
S2 ST signals an RT to dispatch a command block

Figure 5.1) FG-SDS implemetation architecture

Rendering Thread (RT) – An application's thread which needs scheduling to dispatch its command blocks to the GPU. In Figure 5.1, N such threads are shown in the shaded boxes labeled as RT_1 to RT_N. Each RT has to perform five steps, shown as R1 to R5 in the figure, in sequence during its initialization phase before it can enqueue new command blocks in its *GCBQ*. After dispatching its command blocks the first time, every next time during its life time it has to perform steps R2 to R5 in a sequential order each time it wants to dispatch its command blocks to the GPU.

Scheduling Thread (ST) – The thread that runs a scheduling algorithm and, based on the implemented scheduling policy, selects one of the RTs which are waiting for a scheduling decision in the dispatcher. For this purpose, during each scheduling cycle it has to take two steps in general which are labeled as S1 and S2 in the Figure 5.1.

GPU Scheduler Queue (GSQ) – The queue that is written by RTs, one at a time, during their initialization. It is also read by the ST for the identification and selection of an RT during a scheduling decision. See steps R1 and S1 in the Figure 5.1.

GPU Command Block Queue (GCBQ) – Each RT initializes one such queue for itself at the start of scheduling. During the execution it inserts its *GPU Command Blocks (GCBs)* in this queue which are later to be dispatched to the GPU for execution. See step R2 in the Figure 5.1.

Dispatcher – Whenever an RT gets ready to dispatch its commands to the GPU, it gets blocked in the dispatcher, a logical component, until it is selected in a scheduling decision by the ST. Once it is selected, the ST gives it the permission to dispatch its command block which is enqueued earliest in its $GCBQ$. After dispatching its commands to the GPU it acknowledges the ST from within the dispatcher. See steps R3,R4 and R5 of an RT and step S2 of the ST.

The rest of this chapter is organized as follows. In the next subsection we describe the architecture of the queues in detail which are used by the FG-SDS model. The functional behavior of the Rendering Thread and the Scheduling Thread are presented in the next two subsequent subsections respectively.

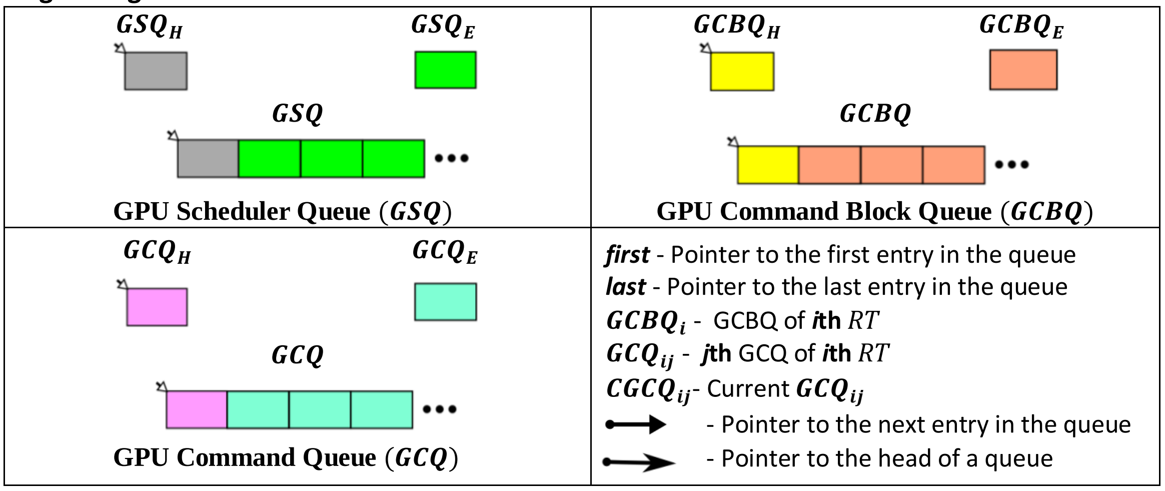
5.2. FG-SDS Queues Architecture

In order to handle commands from multiple OpenGL ES 2.0 applications, three types of queues are managed in the FG-SDS model which are mentioned as follows.

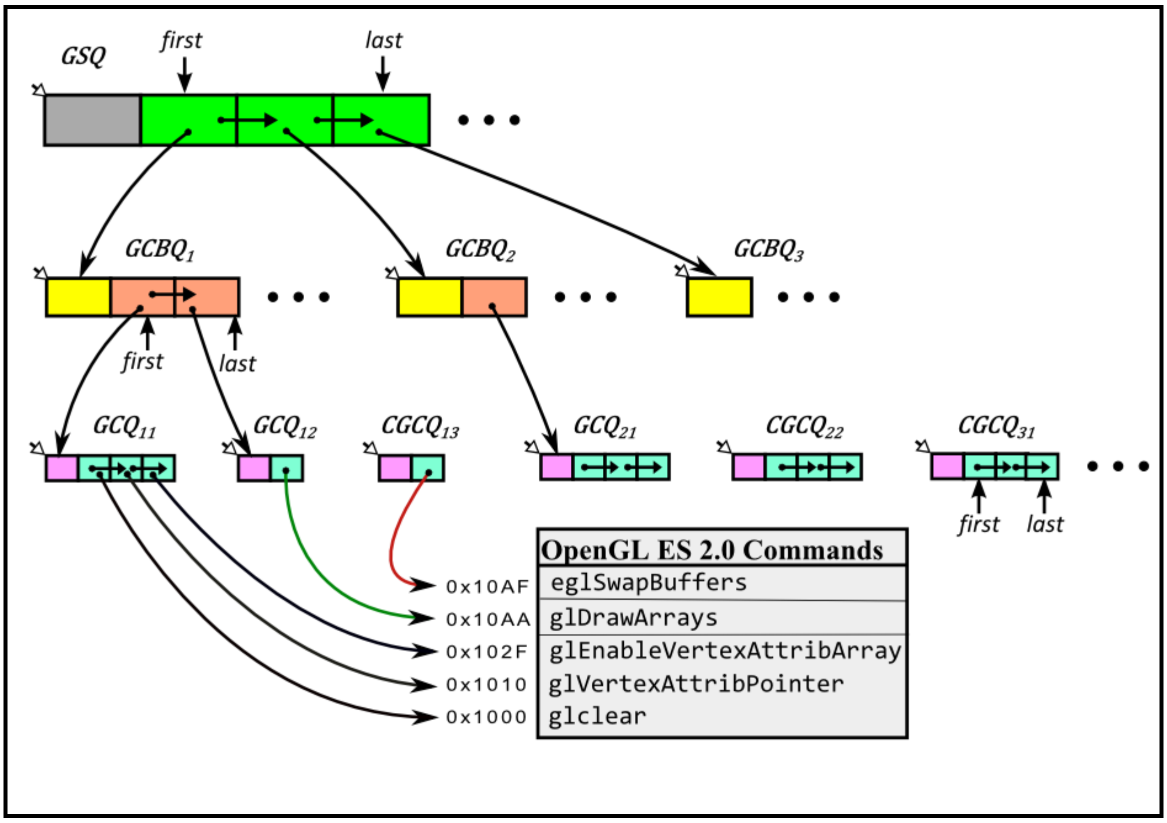
1. GPU Scheduler Queue
2. GPU Command Block Queue
3. GPU Command Queue

For simplicity of making a reference to all these queues we call them FG-SDS Queues. An illustration of the FG-SDS Queues Architecture is shown in Figure 5.2. Each queue in this architecture is composed of two entities, a Queue-Head (Q_H) and a Queue-Entry (Q_E). A Q_H holds all the information required to perform an operation, such as insertion or removal of a Q_E, on the queue and is created whenever a queue is initialized. The queue entries can be dynamically added or removed from a queue. More specifically, a Q_E can be added at the tail of a queue and removed from the head. A detailed description of these queues is given in the following subsections.

Figure Legend:


GSQ_H
GSQ_E
GSQ
GPU Scheduler Queue (GSQ)
$GCBQ_H$
$GCBQ_E$
$GCBQ$
GPU Command Block Queue ($GCBQ$)
GCQ_H
GCQ_E
GCQ
GPU Command Queue (GCQ)

first - Pointer to the first entry in the queue
last - Pointer to the last entry in the queue
$GCBQ_i$ - GCBQ of *i*th RT
GCQ_{ij} - *j*th GCQ of *i*th RT
$CGCQ_{ij}$- Current GCQ_{ij}
- Pointer to the next entry in the queue
- Pointer to the head of a queue



OpenGL ES 2.0 Commands	
0x10AF	eglSwapBuffers
0x10AA	glDrawArrays
0x102F	glEnableVertexAttribArray
0x1010	glVertexAttribPointer
0x1000	glclear

Figure 5.2) FG-SDS Queues Architecture

5.2.1. GPU Scheduler Queue

A GPU Scheduler Queue (GSQ) is a queue which has references to all the $GCBQs$ of the RTs running at a time. In Figure 5.2, it is shown at the top level.

In FG-SDS model there is one such queue in the system. As its name implies, this queue is used by the ST to have access to a RT specific data (stored in its $GCBQ$). The information stored in a GSQ_H and a GSQ_E is listed in the following table.

GSQ_H	GSQ_E
• Pointer to the first GSQ_E • Pointer to the last GSQ_E • A mutex (mutex_st) • A Semaphore (sem_st) • Flag for sorting (flag_sort)	• Pointer to a $GCBQ_H$ • Pointer to the next $GCBQ_E$

Table 5-1) List of parameters in a GSQ_H and a GSQ_E

The pointers listed in the above table are apparent from their description. Since there are multiple RTs which write to GSQ concurrently, a locking mechanism is provided using a mutex variable mutex_st . It is used to grant exclusive access to one of the RTs to the GSQ at a time. Similarly, sem_st is binary semaphore variable which is used by the RTs to synchronize their execution with the ST. An RT sends an acknowledgement to the ST after dispatching its command block by using this variable. The sorting flag flag_sort is used by the ST to perform sorting on its arrays if it is true.

5.2.2. GPU Command Block Queue

A GPU Command Block Queue ($GCBQ$), is a queue of command blocks or GCBs. In Figure 5.2, three queues of this type are shown at the middle level.

There is a 1:1 mapping between the number of RTs and the number of these queues in FG-SDS model. The parameters stored in a $GCBQ_H$ and a $GCBQ_E$ are listed in the table shown below.

$GCBQ_H$	$GCBQ_E$
• Pointer to the first $GCBQ_E$ • Pointer to the last $GCBQ_E$ • A Semaphore (sem_rt) • Wait flag (flag_rt_wait) • Exit flag (flag_rt_exit) • Frame end flag (flag_rt_frame_end) • Dispatch all flag (flag_rt_dispatch_all) • Current frame deadline • Frame counter	• Pointer to a GCQ_H • Pointer to the next $GCBQ_E$

Table 5-2) List of parameters in a $GCBQ_H$ and a $GCBQ_E$

The pointers listed in the Table 5-2 are again clear from the description given. The other parameters are described as follows.

The ST signals an RT to dispatch its commands by using its semaphore variable sem_rt. An RT indicates that it is waiting for the scheduling decision by using flag_rt_wait. Similarly, when an RT needs to exit it sets flag_rt_exit. The end of a frame is indicated by an RT with the help of flag_rt_frame_end. An RT tells the ST whether it wants to dispatch all command blocks or not by using flag_rt_dispatch_all. Current frame deadline of an RT is calculated based on the time of a scheduling decision and the period of a frame. The number of frames dispatched by an RT are also recorded with the help of a frame counter during the execution.

5.2.3. GPU Command Queue

A GPU Command Queue (GCQ), which can also be called as a GPU Command Block (GCB), consists of a block of commands that are to be executed in order and without preemption on the GPU. It is also specified as the minimum scheduling granularity of the FG-SDS model.

These queues are shown at the bottom level in Figure 5.2. The information held by a GCQ-Head (GCQ_H) and GCQ_E is shown in the following table.

GCQ_H	GCQ_E
• Pointer to the first GCQ_E • Pointer to the last GCQ_E • Predicted Execution Time (PET) of the command block	• Pointer to an GPU command • Pointer to the next GCQ_E • GPU command class

Table 5-3) List of parameters in a GCQ_H and a GCQ_E

The PET of a GCB is the time required by all the commands of that GCB to finish their execution on a GPU. The GPU command class represents one of the command classes in FG-SDS model. These command classes are discussed in the next subsection.

5.3. GPU Command Classes

Since there are different kinds of commands with their own semantics and execution requirements, in FG-SDS model we have defined different classes of commands based on their semantics. These Command Classes (*CClasses*), along with their description and an example for each, are listed in the following table.

GPU Command Class	Description	Example
GPU_COMMAND_UNKNOWN	Commands for which a class is not defined	-
GPU_COMMAND_CPUONLY	Commands that are executed on a CPU	`glClear()`
GPU_COMMAND_DRAW	Commands for drawing primitives	`glDrawArrays()`
GPU_COMMAND_MEMCOPY	Commands performing a memory write operation	`glTextImage2D()`
GPU_COMMAND_SYNCHRONOUS	Commands that return something or perform synchronization	`glGetVertexAttribiv()`
GPU_COMMAND_SWAP	Command which change the contents of the framebuffer	`eglSwapBuffers()`

Table 5-4) List of OpenGL ES 2.0 Command Classes, their description and an example in FG-SDS model

5.4. Rendering Thread Context

In order to get the references to its queues and to access other required information quickly, each RT uses a Rendering Thread Context (RTC) to store this information locally. The parameters stored in the RTC of an RT are shown in the following table.

1. A reference to the $GCBQ_H$ associated with the calling RT
2. A reference to the Current $GCQ_H(CGCQ_H)$, in which the commands will be inserted
3. The *CClass* of the command enqueued most recently in the GCQ
4. Application id of the calling RT

Table 5-5) Parameters stored in Rendering Thread Context

The first parameter tells an RT the $GCBQ$ which belongs to it. By using the second parameter an RT identifies the GCQ on which it has to perform an enqueue or a dequeue operation. Since command semantics are used to drive the FG-SDS model, the third parameter tells an RT the *CClass* of the recently enqueued command in the GCQ. The last parameter stores the identity of the application to which the RT belongs.

In the next subsection the steps followed by an RT during its execution are discussed in detail.

5.5. Rendering Thread Functional Behavior

Whenever an RT of an OpenGL ES 2.0 application wants to execute its commands on the GPU it has to follow the steps which are described in sequence in the following subsections.

5.5.1. RT Initialization

Before any of the command blocks are dispatched to the GPU, an RT has to initialize the required resources first. For this purpose it has to call the function shown in the following table.

```
int InitRTresources( $Application\ Id$ )
1.  If Scheduler is not initialized then {
2.          wait until the scheduler is initialized
3.  }
4.  Initialize a $GCBQ_H$
5.  Initialize a $GCQ_H$
6.  Initialize the $RTC$
7.  Store $GCBQ_H$, $GCQ_H$ and $Application\ Id$ in the $RTC$
8.  Initialize a $GSQ_E$ and store $GCBQ_H$ in it
9.  Insert the $GSQ_E$ at the tail of $GSQ$ and set the sorting flag with locking mutex_st
10. Return No_Error_Number
```

Table 5-6) Function called by an RT for initialization of required resources

In the start if the scheduler is not initialized, the calling RT waits until it is initialized. Then it proceeds and performs the steps marked on the lines 4-10 in the above table.

The steps at lines 4, 5 and 6 indicate the resources which are initialized by the calling RT. The step at line 6 initializes the RTC, which is described in section 5.4. In line 7, the first two parameters of the RTC are initialized with the references to the queues initialized at lines 4 and 5 respectively. Moreover, $Application\ Id$ is also stored in RTC. Then a GSQ_E is initialized and a reference to the $GCBQ_H$ is stored in it. It is indicated at line 8. At line 9, the GSQ_E is inserted at the tail of GSQ and the sorting flag is set. This operation is performed with a locking mechanism (locking `mutex_st`), described in section 5.2.1, which allows one of the calling RTs at a time to insert its GSQ_E in the GSQ. If no error occurs during the aforementioned steps, the RT finally returns with a successful initialization (see line 10).

5.5.2. Enqueue Commands

Once an RT has successfully initialized the required resources, it starts enqueuing its commands by calling the function shown in the Table 5-7. The sequence of steps in this function is described as follows.

In step 1, the calling RT gets its RTC. If it is initialized then it proceeds to perform steps mentioned in lines 3-8, otherwise it returns an error number. The step at line 3 indicates the initialization of a command block entry (GCQ_E). Then in the next step it initializes the first frame deadline. Then it stores the function parameters, *GPU Command* and *CClass*, in GCQ_E. Then the calling RT gets the reference to the $CGCQ$ from the RTC (see line 5). This is the current command block in which it has to insert the GCQ_E, which is done in the next step at line 6. Then it stores the command class of the recently enqueued command in its RTC. Finally, it returns successfully if no error occurs in the previous steps.

```
int enqueGPUCommand(GPU Command, CClass)
1.  Get the $RTC$
2.  If $RTC$ is initialized then{
3.          Initialize a $GCQ_E$
4.          Initialize the first frame deadline if not initialize before
5.          Store the $CClass$ and $GPU\ Command$ in $GCQ_E$
6.          Get the reference to the $CGCQ_H$ from $RTC$
7.          Insert the $GCQ_E$ at the tail of $CGCQ$
8.          Store $CClass$ in the $RTC$
9.          Return No_Error_Number
10. }
11. Else {  return Error_Number  }
```

Table 5-7) Function called by an RT for enqueuing commands

This function is called each time an RT wants to enqueue a command. The commands are inserted into the same command block until the RT calls **enqueGPUtimePrediction ( )** function which is discussed next.

5.5.3. Command Block End

After enqueuing all the commands of a command block (GCB), an RT marks the end of it by calling the function shown in the Table 5-8. The steps taken by the calling RT in this function are described as follows.

In the first step, the calling RT gets its RTC. If it is initialized then it proceeds and performs the steps between lines 3 and 13, otherwise it returns an error number. It identifies the current command block in the step at line 3. In the next step, it stores the function parameter, Predicted Execution Time (PET), for the current command block.

At this point the current command block is finished and becomes ready to be inserted in the $GCBQ$ of the calling RT. For this purpose, the RT performs the steps at lines 5-8 and finally the current command block is inserted at the tail of the $GCBQ$.

In the next step (see lines 9 and 10), the calling RT interprets the semantics of the last command in the current command block by using the $CClass$ stored in the RTC. If it is *GPU_COMMAND_SWAP* or *GPU_COMMAND_SYNCHRONOUS* then the RT calls the dispatch function (line 11), which is discussed in the next subsection, before proceeding to the next steps. At this point all the command blocks, which are enqueued by the calling RT so far, are to be dispatched to the GPU before it can continue enqueuing its next commands.

```
int enqueGPUtimePrediction( $PET$ )
1.  Get the $RTC$
2.  If $RTC$ is initialized then {
3.          Get the reference to $CGCQ_H$ from $RTC$
4.          Store the $PET$ in $CGCQ_H$
5.          Initialize a $GCBQ_E$
6.          Store a reference to the $CGCQ_H$ in $GCBQ_E$
7.          Get the reference to $GCBQ$ from $RTC$
8.          Insert the $GCBQ_E$ at the tail of $GCBQ$
9.          If $CClass$ in the $RTC$ is $GPU\_COMMAND\_SWAP$ or
10.         $GPU\_COMMAND\_SYNCHRONOUS$ then {
11.                 Call dispatchCommands ( )
12.         }
13.         Initialize a new $GCQ$ and make it $CGCQ$ in the $RTC$
14.         Return No_Error_Number
15. }Else { return Error_Number }
```

Table 5-8) Function called by an RT at the end of a command block

On the other hand, if the $CClass$ in the RTC of the calling RT is other than the aforementioned two command classes then it proceeds to the step at line 13 without calling dispatch function. In this case the currently enqueued command block ($CGCQ$) remains in the $GCBQ$ and the RT initializes a new command block. Then it makes the newly initialized GCQ as the $CGCQ$ in the RTC. This is the GCQ in which the next commands will be inserted.

Finally, if no error occurs in any of the previous steps the calling RT returns successfully (see line 14).

5.5.4. Dispatch Policy

When an RT wants to dispatch its commands to the GPU it calls the function shown in the Table 5-9 and Table 5-10.

In the start the calling RT gets its RTC. If it is initialized then it proceeds with the dispatch otherwise it returns an error number. Line 2 indicates that it gets the reference to its $GCBQ$ from RTC. Line 3 indicates that it stores the reference of the first entry in its $GCBQ$ in a temporary variable Current_$GCBQ_E$.

Lines 6-11 and 17-22 indicate the two blocking possibilities for the calling RT depending on the status of its flag_rt_dispatch_all flag. In the former case it blocks only in the start while in the latter case it blocks each time before dispatching a command block. However, in both the cases it performs the same steps which are as follows.

int **dispatchCommands( )**

```
Get the $RTC$
If $RTC$ is initialized then
{
        Get the reference to $GCBQ_H$ from $RTC$
        Current_$GCBQ_E$ = First $GCBQ_E$ in $GCBQ$
        If flag_rt_dispatch_all is true then{
                Set flag_rt_wait if not set already
                Wait on sem_rt ($RT$ and $ST$ synchronization point )

                Reset flag_rt_wait if set
        }
        // start dispatching command blocks from $GCBQ$
        While( Current_$GCBQ_E$ is Not Null)
        {
                Current_ $GCQ$ = $GCQ_H$ stored in Current_$GCBQ_E$
                Current_$GCQ_E$ = First $GCQ_E$ in Current_ $GCQ$
                If flag_rt_dispatch_all is false then{
                        Set flag_rt_wait if not set already
                        Wait on sem_rt ($RT$ and $ST$ synchronization point )

                        Reset flag_rt_wait if set
                }

                While( Current_$GCQ_E$ is Not Null){
                        Dispatch command in Current_$GCQ_E$ to GPU
                        Store $CClass$ of the command dispatched in $RTC$
                        Remove $GCQ_E$ from the head of Current_ $GCQ$
                        Current_$GCQ_E$ = First $GCQ_E$ in Current_ $GCQ$
                }
                Remove $GCBQ_E$ from the head of $GCBQ$
                Current_$GCBQ_E$ = First $GCBQ_E$ in $GCBQ$
                If $CClass$ in $RTC$ is $GPU\_COMMAND\_SWAP$ then {
                        Record Current_Time as Frame_Finish_Time
                        Increment frame counter
                        Set flag_rt_frame_end if not set already
                }
                // signal $ST$ here after dispatching a $GCB$ if blocking is on each $GCB$
                If flag_rt_dispatch_all is false then {
                        Dispatch Flush command
                        Set scheduler next activation to Current_Time + Dispatched
                        GCQ predicted time
                        Post sem_st ($RT$ and $ST$ synchronization point )
                }
        } // outer while end
```

(Case -1)
Block only in the start, then dispatch all $GCBs$

(Case – 2)
Block before dispatching each GCB

Dispatch a command block (GCB)

End of a frame

(Case – 2)
Acknowledge the ST

Table 5-9) Function called by an RT for dispatching its command blocks (Part-1)

```
45.           // signal ST here after dispatching all GCBs if block was only in the start
46.           If flag_rt_dispatch_all is true then{
47.                 Set scheduler next activation to                        ( Case – 1 )
48.                 Current_Time + sum of predicted time of all GCBs        Acknowled-
49.                 which are dispatched in current frame                   ge the ST
50.                 Post sem_st (RT and ST synchronization point )
51.           }
52.           // Set Next frame deadline
53.           If CClass in RTC is GPU_COMMAND_SWAP then {
54.                 If Frame_Finish_Time > Current_Frame_Deadline then{
55.                       Reset flag_rt_frame_end
56.                       Current_Frame_Deadline = Current_Time + getPeriod( );
57.                 }else{
58.                       Wait until the Current_Time reaches the
59.                           Current_Frame_Deadline
60.                       Reset flag_rt_frame_end
61.                       Current_Frame_Deadline += getPeriod( )    Set next frame
62.                 }                                               deadline and
63.           }                                                     proceed
64.           Return No_Error_Number
65.   }Else { return an error number }
```

Table 5-10) Function called by an RT for dispatching its command blocks (Part-2)

It sets its wait flag flag_rt_wait to true and then waits on its semaphore to receive a signal from the *ST*. It is indicated at lines 7, 8 and 18, 19 respectively. When it receives a signal from the *ST*, it gets unblocked. Then it resets its wait flag (flag_rt_wait) if it is set before. Lines 10 and 21 indicate these steps.

In order to dispatch its command blocks the calling *RT* iterates through its $GCBQ$ until it finds the last $GCBQ_E$. Line 13 indicates the while loop which checks this condition. Each time it gets a valid $GCBQ_E$, it gets the reference to a GCQ_H from it and stores it in a temporary variable Current_ GCQ (see line 15). Line 16 indicates that it uses another temporary variable Current_GCQ_E to store the first entry in the Current_ GCQ.

Lines 24-29 indicate the while loop in which each command of the Current_ GCQ is dispatched to the GPU. During this process entries are also removed from the head of the Current_ GCQ after dispatching their commands. Moreover, *CClass* of the most recently dispatched command block is also stored in the *RTC*.

After dispatching the commands of the Current_ GCQ, the head entry of the $GCBQ$ is also removed. The next $GCBQ_E$ in the $GCBQ$ becomes both the first entry as well as the Current_$GCBQ_E$. Lines 30 and 31 indicate these steps.

The current frame ends when a command with *CClass* =*GPU_COMMAND_SWAP* is dispatched. Lines 32-36 indicate the checking of this condition. If it is the case then the calling RT records the finish time for the current frame, increments its frame counter and sets its frame end flag.

At this point the calling RT checks its flag_rt_dispatch_all once again (see line 38-43). If it is false then it dispatches a flush command to ensure that all the commands that are sent before are sent to the GPU for execution. It also sets the next activation time for the ST. Then it sends an acknowledgement to the ST and blocks again if there are more command blocks in its $GCBQ$.

On the other hand if flag_rt_dispatch_all is true then it continues dispatching until all the command blocks are dispatched. Then it sets the next activation time for the ST and sends an acknowledgement to it. Line 46-51 indicate this case.

Before continuing to enqueue the next commands the last step the RT performs is to set the frame deadline for the next frame. Lines 53-63 indicate the procedure by which the deadlines are set.

If no error occurs then after dispatching all the command blocks the calling RT returns successfully (see line 64).

5.5.5. RT Exit Policy

When an RT wants to exit, it calls the function shown in the Table 5-11. The description of the steps taken by the calling RT in this function is as follows.

In the start the calling RT gets its RTC, if it is initialized then it proceeds to free the allocated resources otherwise it returns an error number.

In order to free the allocated resources it sets the exit flag in its $GCBQ_H$ and then waits for a signal from ST. Lines 3-5 indicate these steps. When it receives a signal from the ST, it unblocks and removes all the entries (if any) from its $GCBQ$ and frees the memory allocated to those entries one by one (see line 6).

int **destroyGPUCommandQueue( )**

```
1.  Get the $RTC$
2.  If $RTC$ is initialized then {
3.          Get the reference to $GCBQ_H$ from $RTC$
4.          Set flag_rt_exit
5.          Wait on sem_rt ($RT$ and $ST$ synchronization point )
6.          Remove all queue entries from $GCBQ$ and free the memory
7.          Remove the $GSQ_E$ containing the $GCBQ_H$ from $GSQ$ with locking
8.          mutex_st and free the memory
9.          Post sem_st ($RT$ and $ST$ synchronization point )
10.         Destroy $RTC$
11.         Return No_Error_Number
12. }
13. Else { return Error_Number }
```

Table 5-11) Function called by an RT when it exits

After removing all the command blocks, it removes the GSQ_E which contains its $GCBQ_H$ from the GSQ. This operation is performed with taking a lock on GSQ so that no write or read operation can be performed on GSQ until the removal of GSQ_E completes. In this step the memory allocated to the removed GSQ_E is also freed (see lines 7 and 8). After the de-allocation of all the queue resources, the calling RT acknowledges the ST so that it can proceed (See line 9). In the next step at line 10, the RT destroys its RTC. If no error occurs, the RT finally exits successfully (see line 11).

5.6. GPU Scheduler Initialization

The GPU scheduler is initialized at the start of scheduling. Before any RT can proceed, following function has to be called first.

initGPUscheduler( )
1. Initialize a GSQ_H 2. Define the scheduling policy 3. Create the ST which calls the **schedule( )**

Table 5-12) Function called for GPU Scheduler initialization

This function performs three major steps which are listed in the above table. In step 1, a GSQ is initialized. The scheduling policy is defined in step 2. In the next step the ST is created which runs the scheduling algorithm by calling the **schedule ( )** function which is discussed in the next subsection.

5.7. Scheduling Thread Functional Behavior

The ST created during the initialization of the GPU Scheduler, calls the function which is shown in Table 5-13.

Lines 1-2 in Table 5-13 indicate the initialization of an array of the constant size. The size basically represents the number of possible RTs, which is set before the compilation. The array ($array\_GCBQ\_heads$) is used to store the references to the $GCBQs$ of the RTs. In the start, all the locations of this array are initialized with the NULL values.

The ST then enters an infinite loop to run the scheduling algorithm. In the beginning it checks the GSQ status. It no RT has yet initialized its $GCBQ$ then it waits and continues checking this condition again (see lines 7-9). This process continues until an RT initializes its $GCBQ$ and inserts an entry into the GSQ.

At this point, if there is currently a single entry in the GSQ then the ST takes the steps indicated in the lines 11-17. This block of code reduces the scheduling overhead if there is only one RT in the system at a time. In this block two flags of an RT are checked (see line 14). If any of these flags are set by the RT, it will be selected by the ST.

Lines 29-38 indicate the sending of a signal by the ST to the selected RT and receiving of an acknowledgement from the corresponding RT. After receiving the acknowledgement the ST checks its timing conditions for the next activation. If these conditions are met then it can proceed for the next scheduling decision.

schedule ( )

```
Define constant $array\_size$ // Maximum number of $RTs$
Initialize $array\_GCBQ\_heads[array\_size]$ with each value = NULL

While (1)
{
        Selected_$GCBQ$ = NULL
```

No RT has initialized its $GCBQ$

```
        If $GSQ$ is empty then {
                sleep for 1 micro second and then continue
        }
```

Only one RT has initialized its $GCBQ$

```
        If there is only one $GSQ_E$ in $GSQ$ then
        {
                Get $GCBQ_H$ from the first $GSQ_E$ in $GSQ$
                If flag_rt_exit or flag_rt_wait in $GCBQ_H$ is true then {
                        Selected_$GCBQ$ = $GCBQ_H$
                }
        }
```

More than one RTs have initialized their $GCBQs$

```
        If there are more than one $GSQ_E$ in $GSQ$ then
        {
                If flag_sort in $GSQ$ is true then{
                        Call function sortUsingRTpriorities($array\_GCBQ\_heads$,
                          , $array\_size$)
                }
                Selected_$GCBQ$ = selectRTusingHPFpolicy($array\_GCBQ\_heads$,
                 $array\_size$)
        }
        // If Selected_$GCBQ$ is a valid pointer then signal the corresponding $RT$
        If Selected_$GCBQ$ is not NULL then{
                Post sem_rt of the Selected_$GCBQ$ ($RT$ and $ST$ synchronization
                point )

                Wait on sem_st ($RT$ and $ST$ synchronization point )
                If Current_Time < Next_Activation_Time then {
                        Sleep for (Next_Activation_Time – 1 micro second) if it is
                        Not negative
                }
        }
} // end while loop
```

Signal RT to dispatch or exit, then after receiving acknowledgement proceed if the timing conditions are fulfilled

Table 5-13) Function called by the ST

While on the other side, if more than one RTs insert their entries in $GCBQ$ then the steps mentioned in lines 19-27 are taken. In these steps two functions are called. The first function sorts the references of the $GCBQs$ according to their priorities in $array\_GCBQ\_heads$. The second function selects one of the RTs based on the Highest Priority First (HPF) scheduling policy. Both of these functions are described in the upcoming subsections.

If a selection in made in the function shown at lines 25 and 26, then the corresponding RT is signaled by the ST to dispatch its command blocks (See lines 29-38). In this way RTs are selected by the ST and are signaled to dispatch accordingly.

5.7.1. Sorting Based on RT Priorities

Whenever a new entry is inserted in to the GSQ by an RT, it sets the sorting flag (see line 9 in Table 5-6). When the ST sees this flag being set while running the schedule function, as discussed in the previous subsection (see lines 21-24 in Table 5-13), it calls the function shown in the following table.

```
int sortUsingRTpriorities($array\_GCBQ\_heads$, $array\_size$)
1.  Reset flag_sort with locking mutex_st
2.  Current_$GSQ_E$ = First $GSQ_E$ in $GSQ$
3.  While ( Current_$GSQ_E$ is Not Null){
4.          Get $GCBQ_H$ from Current_$GSQ_E$
5.          $priority$ = Get priority of the $RT$ with $GCBQ_H$
6.          $array\_index = priority - 1$
7.          If $array\_index$ is negative or greater than ( $array\_size - 1$) then{
8.                  Return Error_Number
9.          }
10.         $array\_GCBQ\_heads$ [ $priority - 1$ ] = $GCBQ_H$
11.         Current_$GSQ_E$ = Next $GSQ_E$ in the $GSQ$
12. }
13. Return No_Error_Number
```

Table 5-14) Function called by the ST for sorting RTs accoring to their priorities

This function iterates through the GSQ and sorts the references to the $GCBQs$ in the $array\_GCBQ\_heads$ in descending order of their priorities. This function assumes that each of the RTs has a unique priority number from 1 to $array\_size$ where 1 represents the highest priority and $array\_size$ represents the lowest priority.

In the start, the ST resets the sorting flag with a locking mechanism (see line 1). Then it iterates through the GSQ (see lines 3-12). Each time it gets the $GCBQ_H$ from the Current_GSQ_E and then gets the associated $RT's$ priority as shown at line 5. If the priority number is in the valid range then it places the GSQ_H in $array\_GCBQ\_heads$ as mentioned at line 10. This process continues until all the $GCBQ$s are being sorted. If no error occurs, the ST finally returns successfully with $array\_GCBQ\_heads$ being sorted.

5.7.2. Scheduling Policy

When more than one RTs insert their $GCBQ$ entries into the GSQ then for the selection of an RT, the ST calls the function which is shown in Table 5-15.

Line 1 and 2 indicate some of the variables which are initialized. LP_GCBQ and HP_GCBQ indicate the lower priority and the highest priority $GCBQs$ respectively. The ST starts iterating through the array,$array\_GCBQ\_heads$, until it finds a valid $GCBQ$. The first valid index is where the highest priority $RT's$ $GCBQ$ is located. Lines 13-16 indicate the identification of the highest priority RT. Lines 18-21 indicate that when the wait flag of the highest priority RT is true, it gets selected.

If the highest priority RT is not ready, the ST proceeds and checks the wait flags of the other RTs (see line 22 and 23). If a low priority $RT's$ wait flag is true, then it proceeds to the next step and checks whether the highest priority RT has enqueued a swap command or not. If it is not the case then it continues with the next $GCBQ$ of the next low priority RT in the array.

On the other hand if RT has enqueued a swap command then ST checks whether the highest priority application has finished its frame or not. If the frame end flag of the highest priority application is true then the timing conditions are checked in such a way that if a low priority RT is dispatched it should not cross the deadline of the highest priority RT (See lines 28-40).

In each iteration the highest priority RT gets a chance to be selected if it is ready.

$GCBQ_H$ **selectRTusingHPFpolicy** ($array\_GCBQ\_heads$, $array\_size$)

```
1.  $array\_index$ = 0, $LP\_GCBQ$ = NULL , $HP\_GCBQ$ = NULL
2.  $HP\_GCBQ\_Flag$ = False, $hp\_index$ = 0
3.  While( $array\_index < array\_size$)
4.  {
5.          $Current\_GCBQ_H = array\_GCBQ\_heads$ [ $array\_index$ ]
6.          If $Current\_GCBQ_H$ is not a valid $GCBQ_H$ then{
7.                  Increment $array\_index$ and continue the loop
8.          }
9.          If flag_rt_exit of $Current\_GCBQ_H$ is True then {
10.                 $array\_GCBQ\_heads$ [ $array\_index$ ] = NULL
11.                 Return $Current\_GCBQ_H$
12.         }
13.         If $HP\_GCBQ\_Flag$ = False then{
14.                 $HP\_GCBQ = Current\_GCBQ_H$
15.                 $hp\_index = array\_index$
16.         }
17.
18.         If flag_rt_wait of $HP\_GCBQ$ is true the {
19.                 Set flag_rt_dispatch_all of $HP\_GCBQ$
20.                 Return $HP\_GCBQ$
21.         }
22.         If flag_rt_wait of $Current\_GCBQ_H$ is True and $array\_index$ is not
23.         $hp\_index$ then {
24.                 If $RT$ with $HP\_GCBQ$ has not enqueued a command of
25.                 $GPU\_COMMAND\_SWAP$ command class yet then{
26.                         Increment $array\_index$ and continue the loop
27.                 }
28.                 else{
29.                         $LP\_GCBQ = Current\_GCBQ_H$
30.                         $Current\_GCQ\_Pred\_Time$ = predicted time of the $GCQ$ at
31.                         the head of $Current\_GCQ$
32.                         If flag_rt_frame_end of RT with $HP\_GCBQ$ is True{
33.                                 If (Current Time + $Current\_GCQ\_Pred\_Time$)
34.                                 exceeds the Current Frame Deadline of RT
35.                                 with $HP\_GCBQ$ then{
36.                                         Increment $array\_index$ and continue
37.                                 }
38.                                 Return $LP\_GCBQ$
39.                         }
40.                 }
41.         }
42. }
```

Identification of the Highest Priority RT

Selection of the Highest Priority RT

Selection of a Low Priority RT

Table 5-15) Function which implements the Highest Priority First (HPF) scheduling policy

6. EVALUATION AND RESULTS

In the previous chapter we discussed the implementation details of the FG-SDS model. We also discussed the HPF scheduling policy that is used to grant permission to the RTs to dispatch command blocks to the GPU using FG-SDS model.

This chapter gives the description of, the system specifications on which the evaluation is performed, the multitasking scenarios which are chosen for evaluation, the evaluation metrics that is used to reach a conclusion about the performance of the FG-SDS model and the results which are obtained based on the evaluation metrics.

6.1. System Specifications

Table 6-1 lists the important specifications [31] of the embedded platform i.MX6 [31] by Freescale [32] on which the evaluation is performed.

CPU	4x ARM® Cortex™-A9 up to 1.2 GHz per core
GPU 3D	Vivante GC2000 200Mtri/s 1000Mpxl/s, OpenGL ES 2.0 & Halti, CL EP
GPU 2D(Vector Graphics)	Vivante GC355 300Mpxl/s, OpenVG 1.1
GPU 2D(Composition)	Vivante GC320 600Mpxl/s, BLIT
Operating System	Linux (Ubuntu)

Table 6-1) An overview of the system specification of the i.MX6 embedded platform [31]

One of the important target applications of this platform is the telematics in the automotive industry. As our multitasking scenarios are from telematics, we have chosen this platform for our evaluation. These scenarios are described in the following subsection.

6.2. Multitasking Scenarios

In the future car example that we discussed in section 1.1, we highlighted some of the 3D rendering applications which are executed on a GPU. Our multitasking scenarios are built by some specific applications, such as Speedometer, Tachometer etc., from this application domain.

We have chosen two multitasking scenarios for the evaluation of FG-SDS model and the HPF scheduling policy which are described in the next subsections.

6.2.1. Multitasking Scenario 1

The 3D applications which are included in the Multitasking Scenario 1 (MTS-1) are listed in the following table.

Application No./Priority	Application Name	Desired Frame Rate (FPS)	Period (ms)
1	Speedometer	30	33.33
2	Tachometer	30	33.33
3	Menu	30	33.33

Table 6-2) MTS-1 application set with some QoS parameters

In this scenario we have three applications. The priority, desired frame rate and the frame period for each application is specified and listed in the table above. The Speedometer is a time critical application with a highest priority therefore it has to meet frame deadlines besides maintaining the desired frame rate. The Tachometer is given a medium priority and the third application is a display Menu which is given the lowest priority. Our goal is to guarantee the frame deadlines for the time critical applications and to allocate the GPU to the lower priority applications only if it is feasible.

In this scenario we have chosen the desired frame rates in such a way that the schedule is feasible and the GPU is not overloaded. Therefore all the three applications can maintain the desired frame rates without our scheduling scheme. The order of execution then depends only on the implementation of the driver.

6.2.2. Multitasking Scenario 2

The applications in the Multitasking Scenario 2 (MTS-2) are listed in the following table.

Application No./Priority	Application Name	Desired Frame Rate (FPS)	Period (ms)
1	Speedometer	30	33.33
2	Tachometer	30	33.33
3	Menu	20	50
4	Spam Application	20	50

Table 6-3) MTS-1 application set with some QoS parameters

As we can see, in MTS-2 the first three applications are same as they are in MTS-1; however, in MTS-2 we have added a fourth application as well. This applicaion is a spam application which overloads the GPU. Moreover we have changed the frame rate for the Menu application from 30 FPS to 20 FPS to create some room for the low priority applications to execute.

To measure the performance of the aforementioned scenarios, the evaluation metrics that we used is described in the following subsection.

6.3. Evaluation Metrics

There the three key parameters which define our evaluation metrics for an application. These are the Average Frame Rate (AFR), the Probability of Meeting Deadlines (PMD) and the Probability Distribution of Frame Finish Times (PD-FFT) with respect to their deadlines. These parameters are also somehow interrelated to each other but each of them gives its own measure of the performance of an application in a multitasking scenario. The way in which these parameters are calculated is briefly described below.

6.3.1. Average Frame Rate (AFR)

The AFR of an application is calculated by the following equation.

$$AFR = \frac{Total\ Frames}{End\ Time - Start\ Time} \qquad \dots\ eq.(6.1)$$

6.3.2. Probability of Meeting Deadlines (PMD)

The PMD of an application is calculated by the following equation.

$$PMD = \frac{No.of\ Frames\ which\ met\ Deadlines}{Total\ Frames} \qquad \dots eq.(6.2)$$

6.3.3. Probability Distribution of Frame Finish Times (PD-FFT)

The PD-FFT of an application gives us the details of when exactly an application has finished its frames with respect to the deadlines.

6.4. Results

Before we present the results for both MTS-1 and MTS-2, we discuss the ways in which the results of the FG-SDS (HPF) scheme are compared with the other schemes. Both of these scenarios are executed in three ways which are:

1. Execution without any scheduling, in this case the driver makes the decision about the granting permission to an application for execution on the GPU. In the following discussion of this chapter we will refer to this case as Without Scheduling.
2. Execution with the Frame Rate Restricted Scheduling (FRRS). In this scheme each application cannot exceed the desired frame rate. However, there is no scheduler involved in this scheme. Each application, if it finishes its frame before the deadline, has to wait for its next activation time which depends on its period and the finish time of the current frame.
3. Execution with our FG-SDS (HPF) scheme.

The comparisons are made between the performance of FRRS and the FG-SDS (HPF) schemes, as we can enforce the frame deadlines in both of these cases. However, the case in which no scheduling is applied at all is also included to get an idea of the frame rates that we obtain in a multitasking scenario.

For each of the 3 cases discussed above, both in MTS-1 and MTS-2, applications were allowed to execute for at least 5 minutes so that we have enough samples to record the data for the evaluation. In the next subsection we present the results of our evaluation for MTS-1 and MTS-s.

6.4.1. MTS-1

6.4.1.1. Average Frame Rates

Without Scheduling - we get frame rates as shown in Figure 6.1 for each application. Although in this scenario AFRs are well above the desired level, however, as the number of applications increase or the GPU is overloaded the time critical or higher priority applications get less chance to execute which results in their AFR degradation and missed frame deadlines.

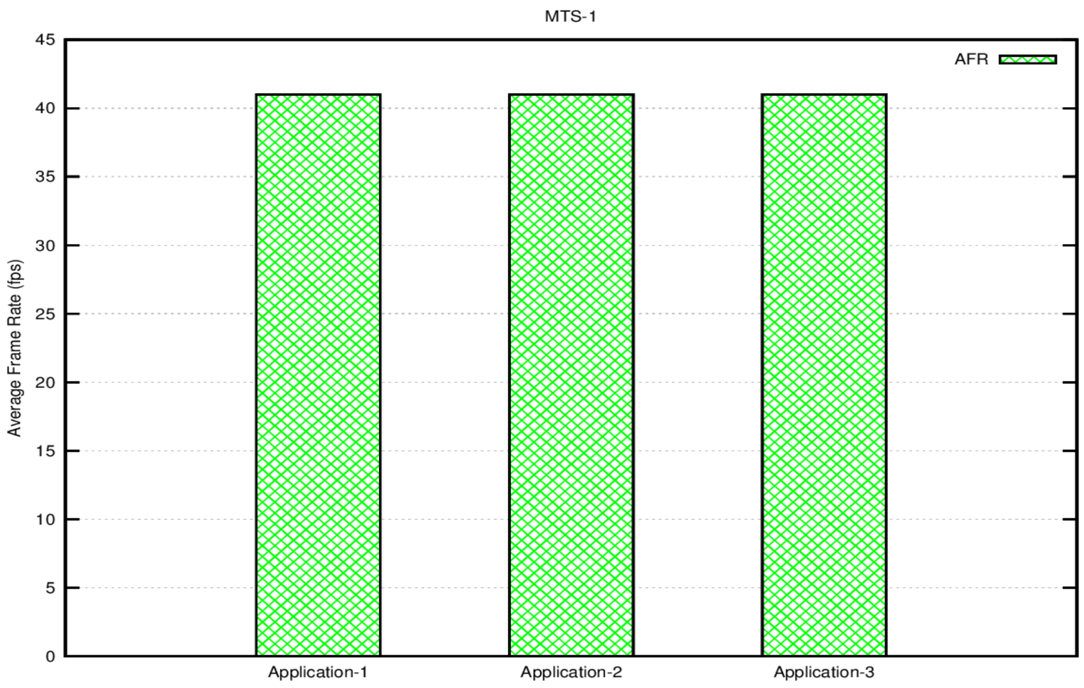


Figure 6.1) AFRs of applications without any scheduling

With FRRS – as expected due to the feasibility of the schedule, each of the applications is able to maintain the desired frame rate as shown in Figure 6.2. However, when the GPU is overloaded we get a different behavior as we will see in MTS-2.

With FG-SDS (HPF) - the highest priority guarantees its desired frame rate which costs in the degradation of AFR of lowest priority application, as shown in Figure 6.2. The reason behind this kind of behavior is because of the scheduling overhead introduced by FG-SDS model.

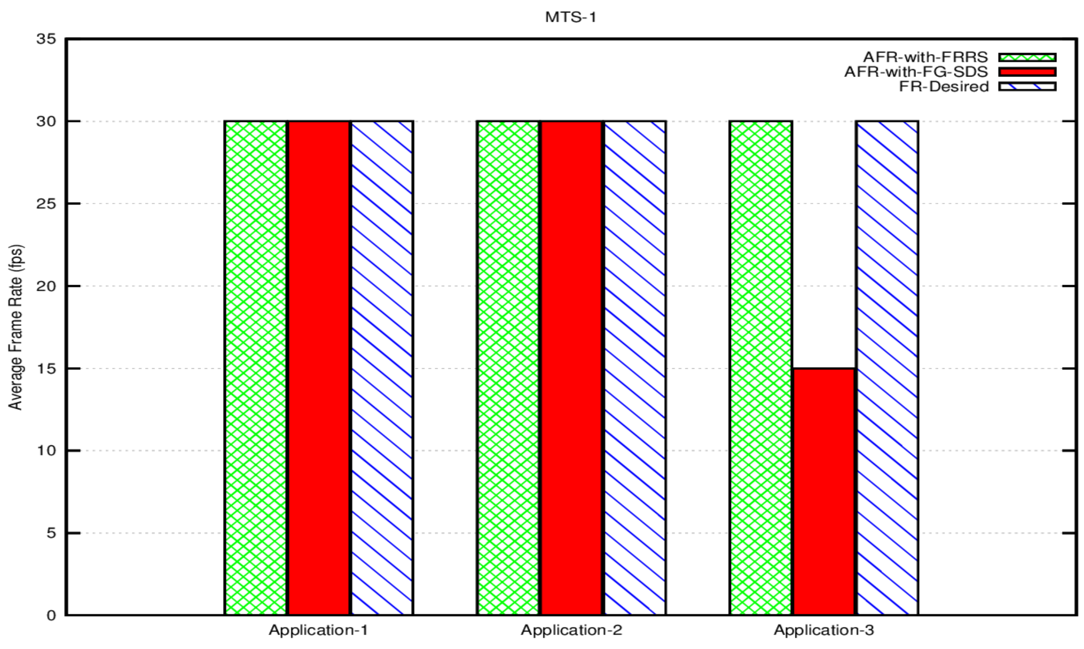


Figure 6.2) A comparison of AFRs of applications with FRRS and FG-SDS (HPF) scheduling schemes

6.4.1.2. Probability of Meeting Deadlines

With FRRS – all three applications are able to meet their frame deadlines, as shown in Figure 6.3, due to the feasibility of the schedule.

With FG-SDS (HPF) - the frame deadlines of the highest priority application are guaranteed which costs in less chance for the low priority applications to meet their deadlines. See the following figure.

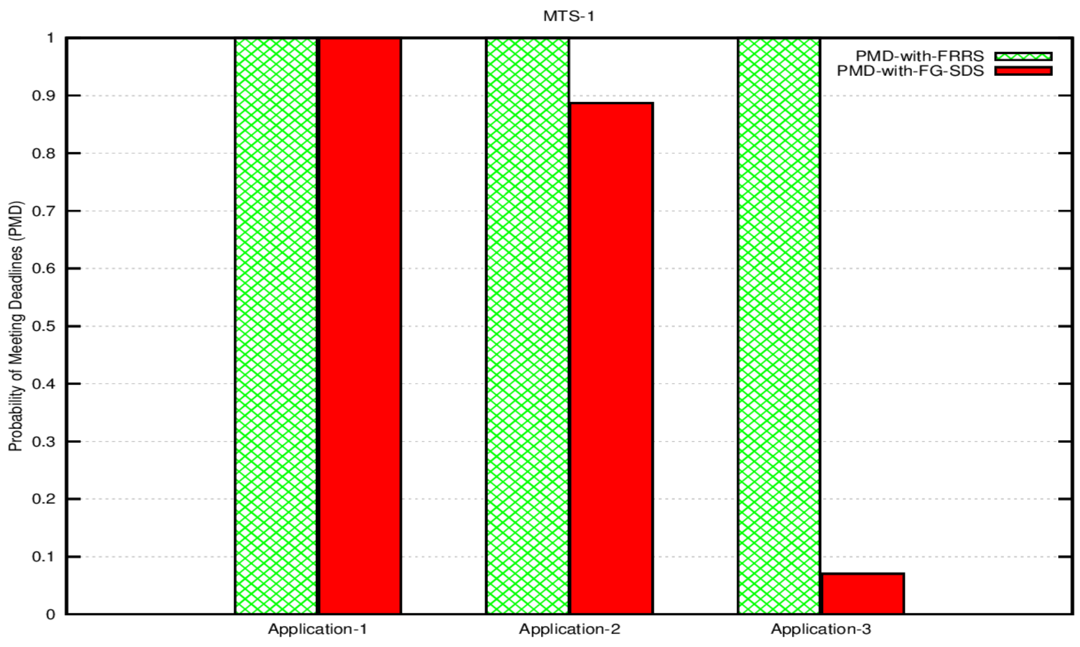


Figure 6.3) A comparison of PMDs of applications with FRRS and FG-SDS (HPF) schemes

6.4.1.3. Probability Distribution of Frame Finish Times

Figure 6.4 and Figure 6.5 represent the PD-FFT of the Application 1 for the FRRS and the FG-SDS (HPF) schemes respectively.

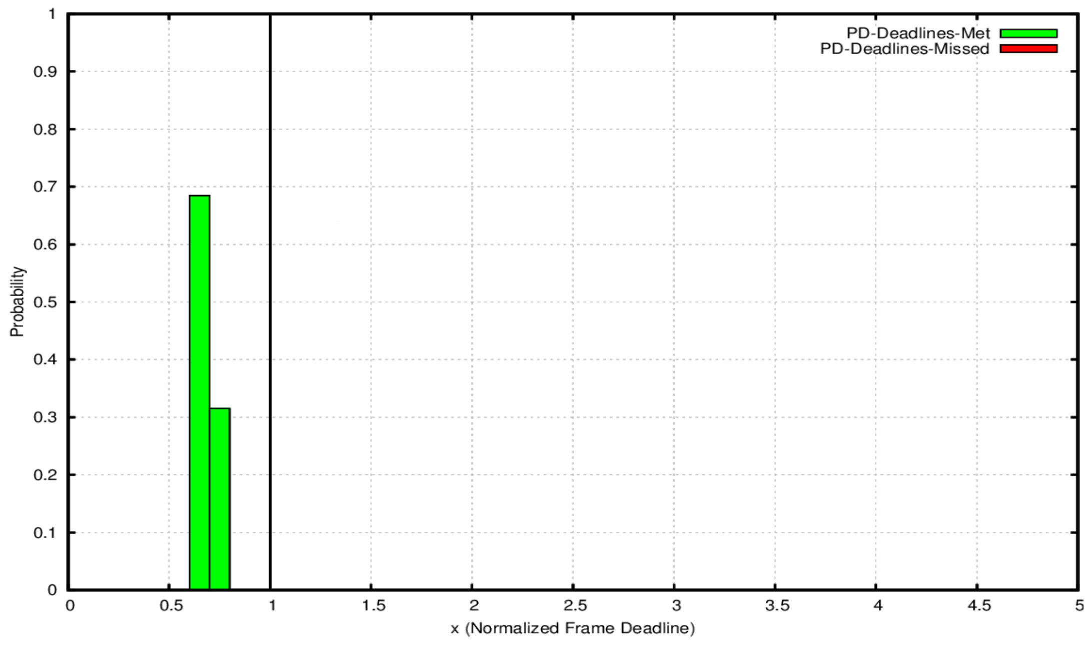


Figure 6.4) Application 1, PD-FFTs using FRRS scheme

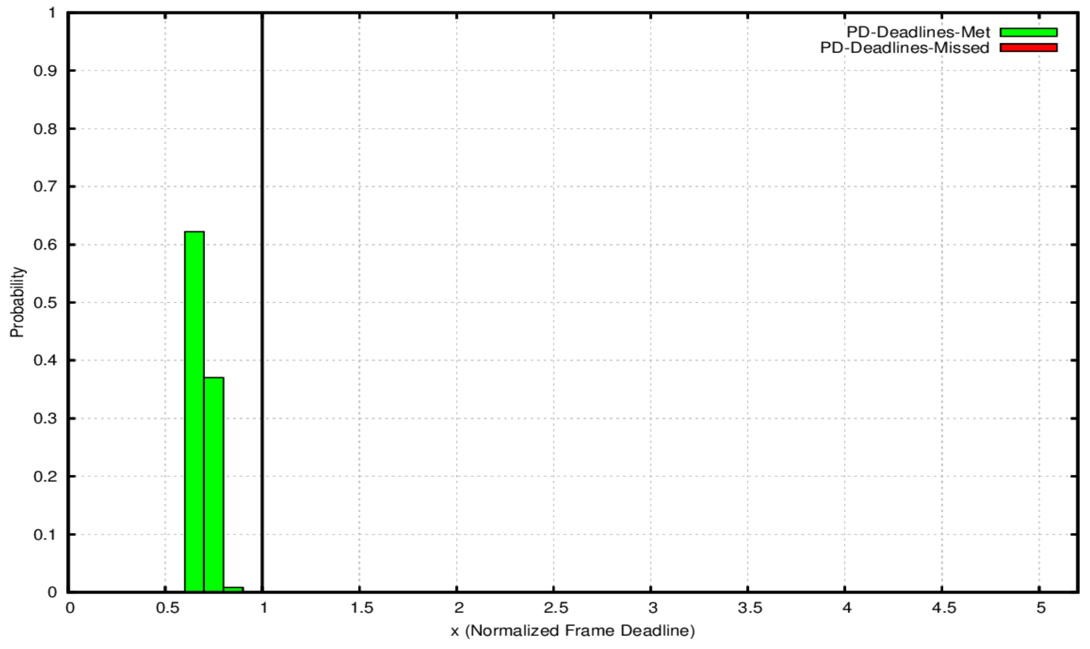


Figure 6.5) Application 1, PD-FFTs using FG-SDS (HPF) scheme

In both cases this application has finished its frames well before its deadline. However, in the former case there is no guarantee that it will do so in a different scenario where the GPU is overloaded.

For the Application 2 the PD-FFTs are shown in Figure 6.6 and Figure 6.7, when FRRS and the FG-SDS (HPF) schemes are applied respectively.

In case of FRRS, again all the frames have finished before their deadlines. However, in the second case some of the frames have missed their deadlines due to the reasons as discussed earlier.

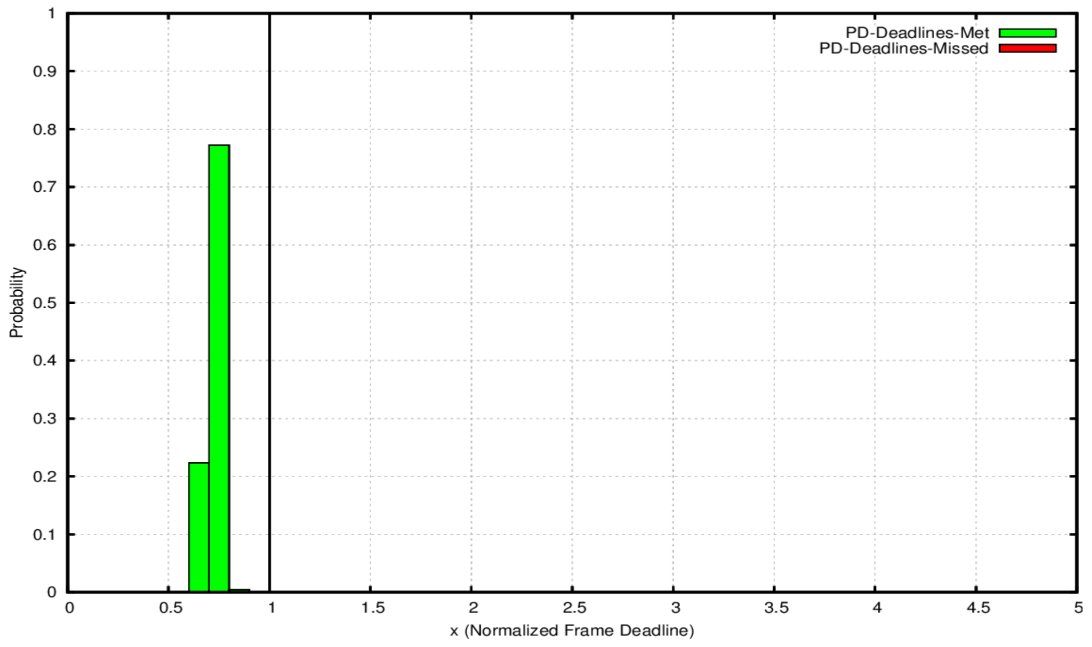


Figure 6.6) Application 2, PD-FFTs using FRRS scheme

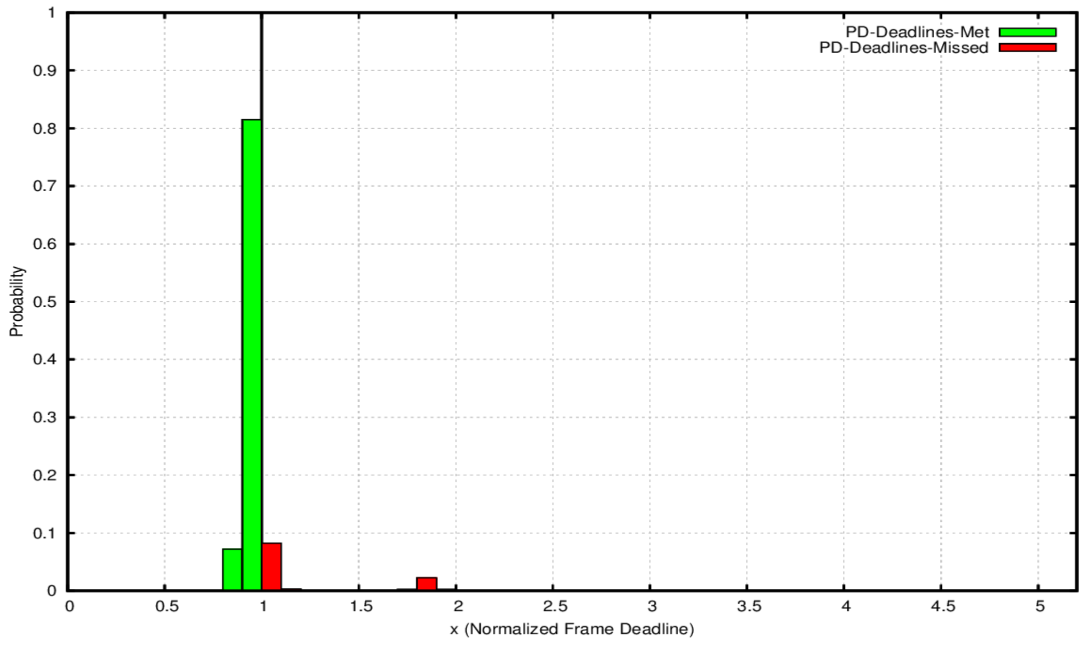


Figure 6.7) Application 2, PD-FFTs using FG-SDS (HPF) scheme

For Application 3, in case of FRRS scheme all the frames have finished before their deadlines as shown in the following figure.

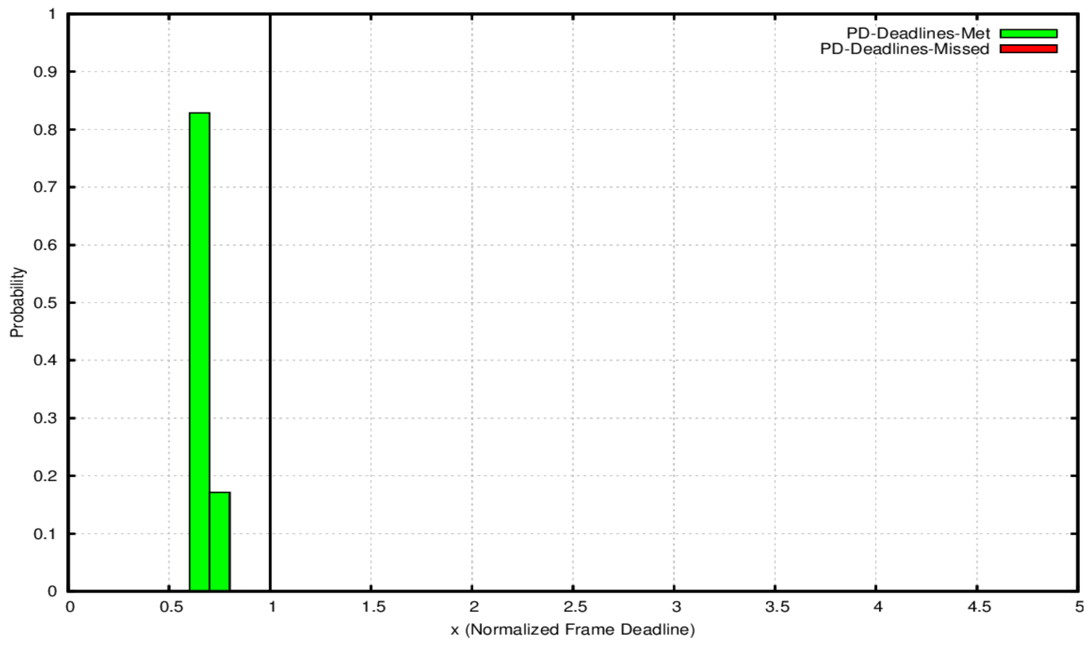


Figure 6.8) Application 3, PD-FFTs using FRRS scheme

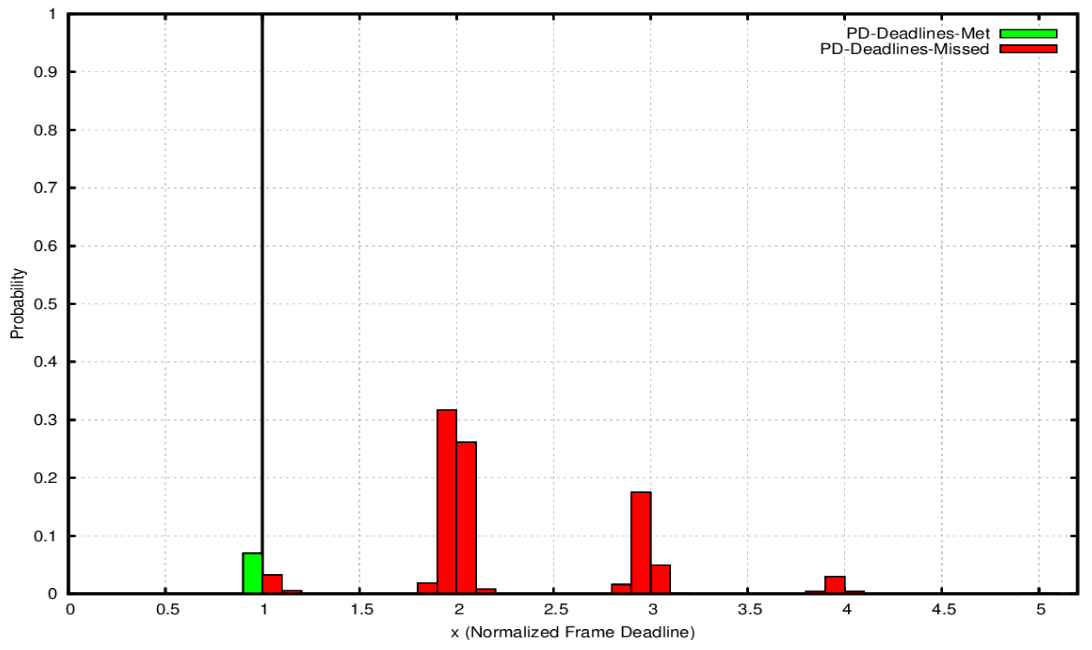


Figure 6.9) Application 3, PD-FFTs using FG-SDS (HPF) scheme

In case of FG-SDS (HPF) scheme, most of the frames have missed their deadlines and get delayed as shown in the figure above.

6.4.2. MTS-2

6.4.2.1. Average Frame Rates

Without Scheduling – all the applications are given the same share of the GPU resources. The frame rates of the applications in this scenario are shown in Figure 6.10. The highest priority application's frame rate dropped significantly from 30 FPS to only 10 FPS. There is no prioritization and all the applications are treated in the same manner.

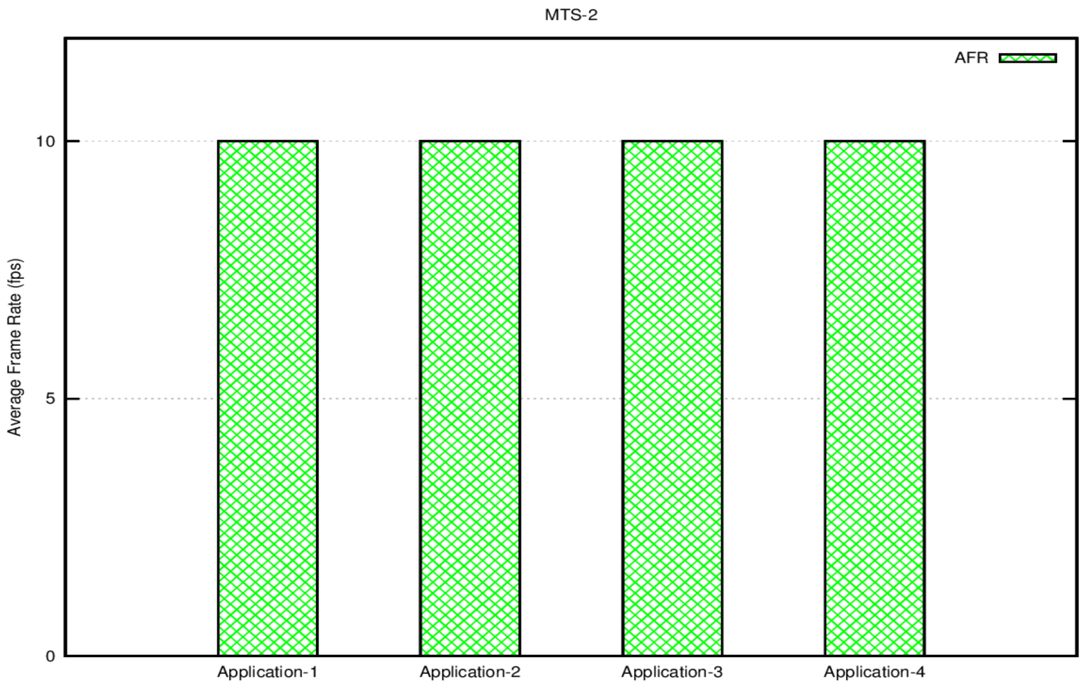


Figure 6.10) AFRs of applications without any scheduling

With FRRS – the applications are treated in the same way as in case of Without Scheduling. From Figure 6.11, we can see that all four applications maintained an AFR of 10 FPS. Clearly there is no prioritization therefore the highest priority application is not able to maintain its desired frame rate.

With FG-SDS (HPF) – it can be clearly seen in Figure 6.11 that the highest priority application maintained its frame rate at the desired level although the GPU is overloaded. The Application 2 and 3 are also able to render according to their priorities therefore the former has a high rate as compared to the latter one. The spam application is not able to render a single frame in this case.

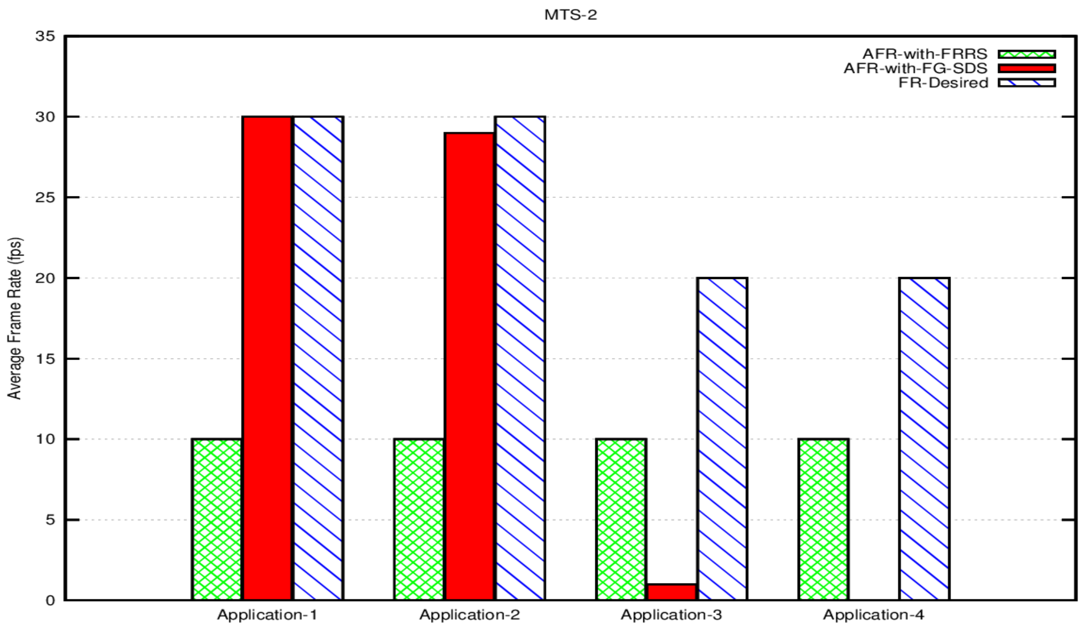


Figure 6.11) A comparison of AFRs of applications with FRRS and FG-SDS (HPF) scheduling schemes

6.4.2.2. Probability of Meeting Deadlines

With FRRS – there is very low probability that the applications, especially the highest priority application, meet their frame deadlines as shown in Figure 6.12.

With FG-SDS (HPF) – the highest priority application meets the frame deadlines all the time as in this case shown in the following figure. Application 2 is also able to meet the deadlines. Only the low priority applications missed their deadlines.

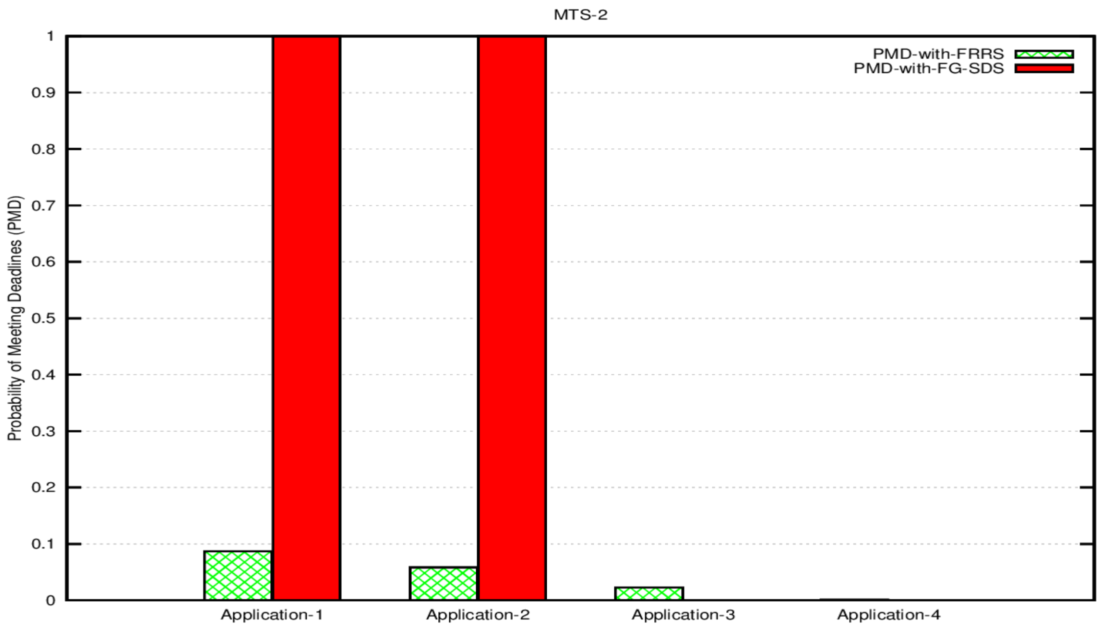


Figure 6.12) A comparison of PMDs of applications with FRRS and FG-SDS (HPF) schemes

6.4.2.3. Probability Distribution of Frame Finish Times

Figure 6.13 shows the PD-FFT of the highest priority application. Only few of the frames which are rendered by this application using FRRS scheme are able to meet their deadlines. On the x scale in this figure all those frames which are delayed by more than a factor of 5 are considered in the bar which appears just after 5.

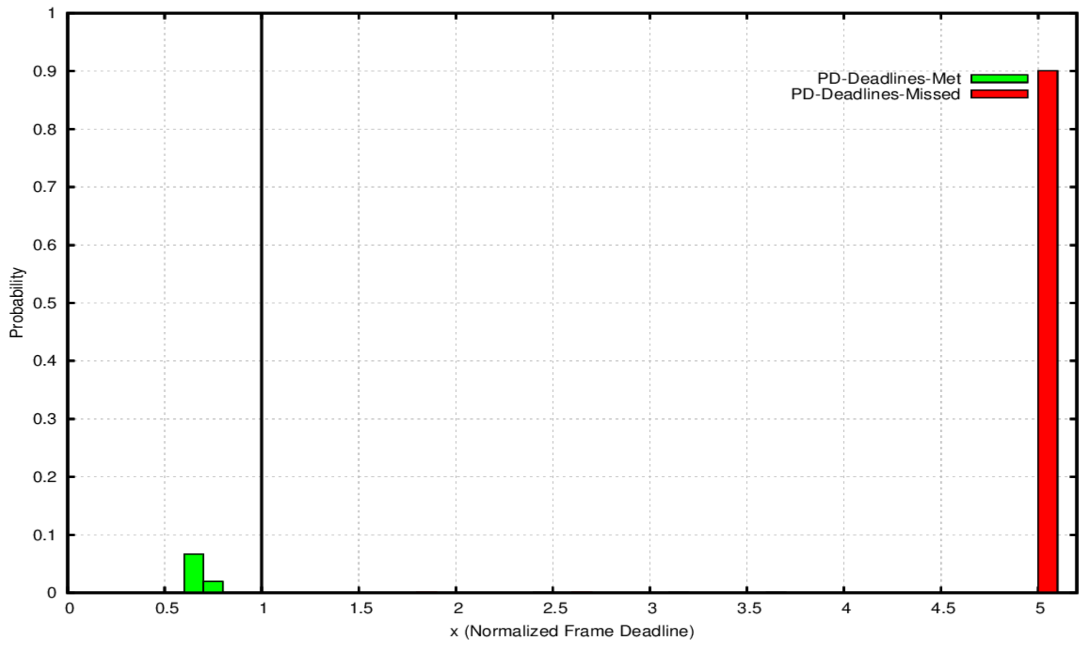


Figure 6.13) Application 1, PD-FFTs using FRRS scheme

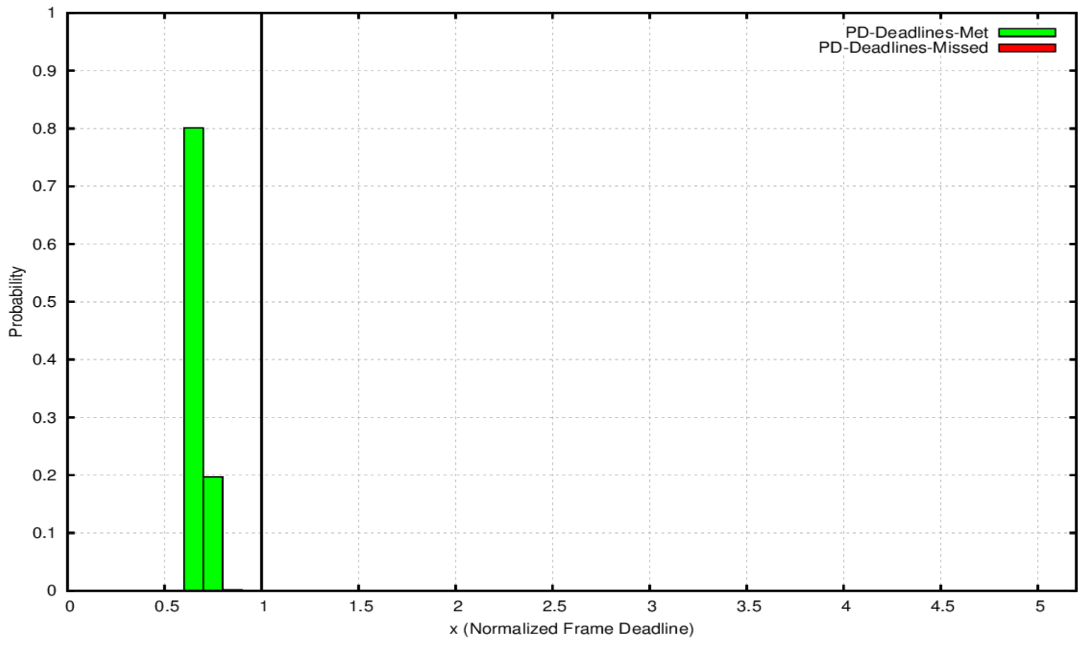


Figure 6.14) Application 1, PD-FFTs using FG-SDS (HPF) scheme

On the other hand, this application is able to render all its frames before their deadlines using FG-SDS (HPF) as shown in the figure above.

Similarly for Application 2, the difference between the results of FRRS and the FG-SDS schemes can been seen in Figure 6.15 and Figure 6.16 respectively.

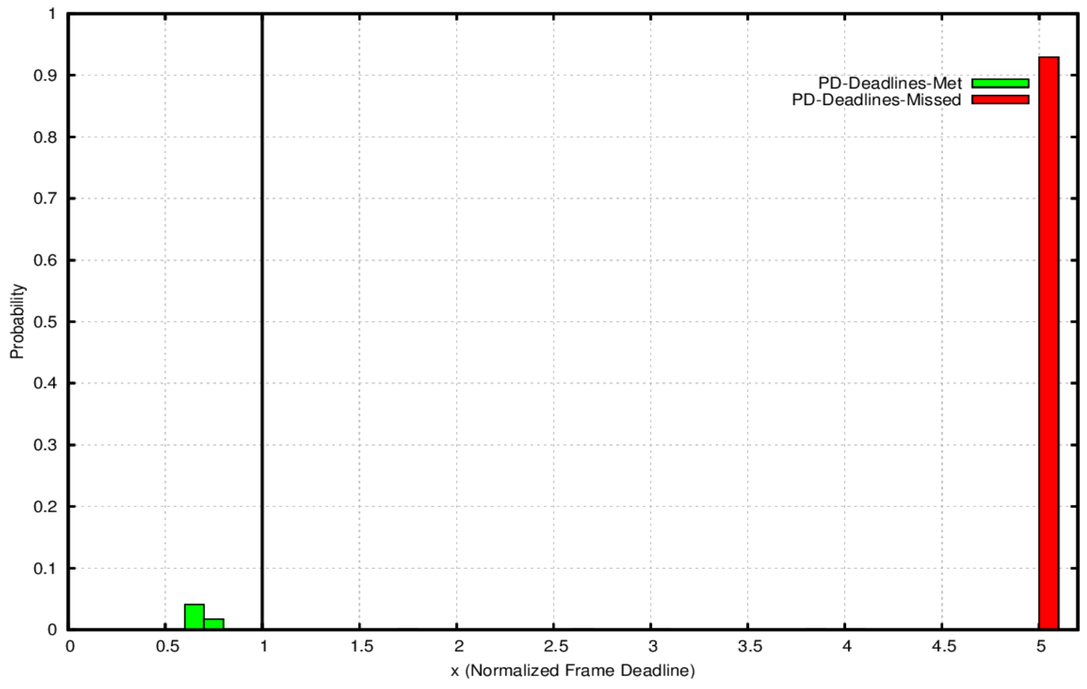


Figure 6.15) Application 2, PD-FFTs using FRRS scheme

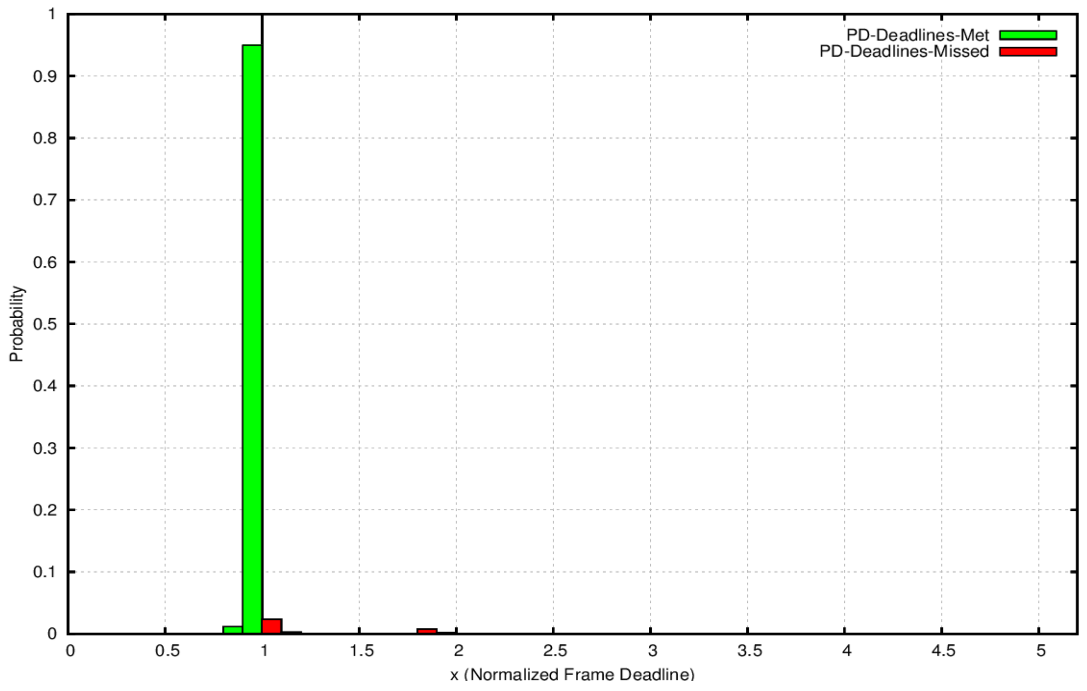


Figure 6.16) Application 2, PD-FFTs using FG-SDS (HPF) scheme

The results for Application 3 are shown in Figure 6.17 and Figure 6.18. Finally the results for Application 4 are shown in Figure 6.19 and Figure 6.20.

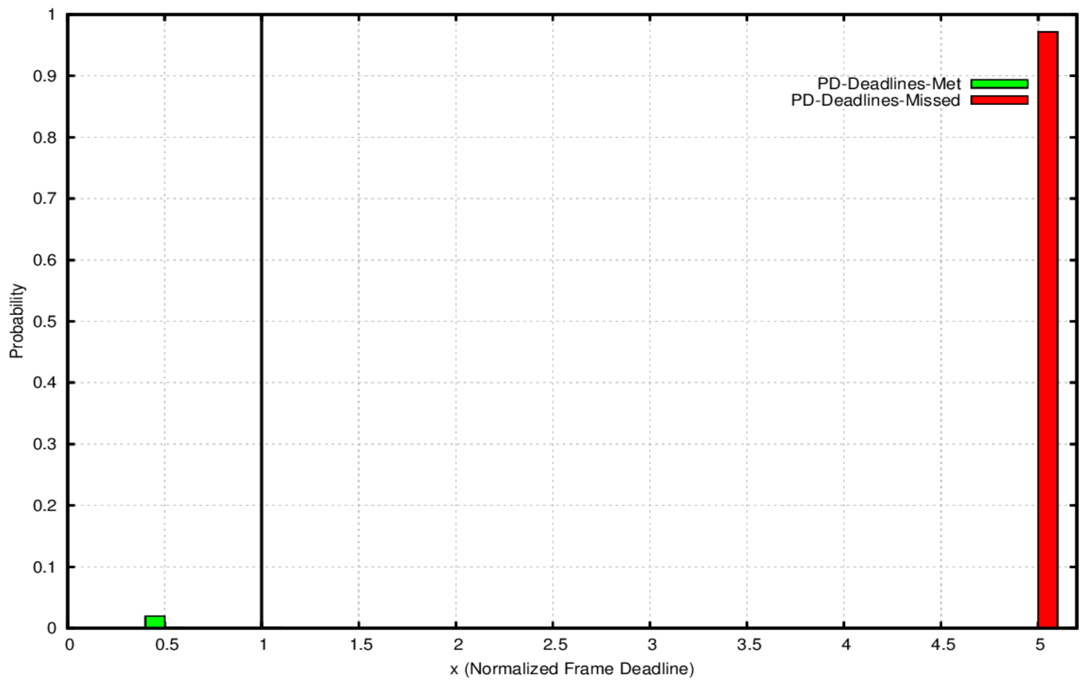


Figure 6.17) Application 3, PD-FFTs using FRRS scheme

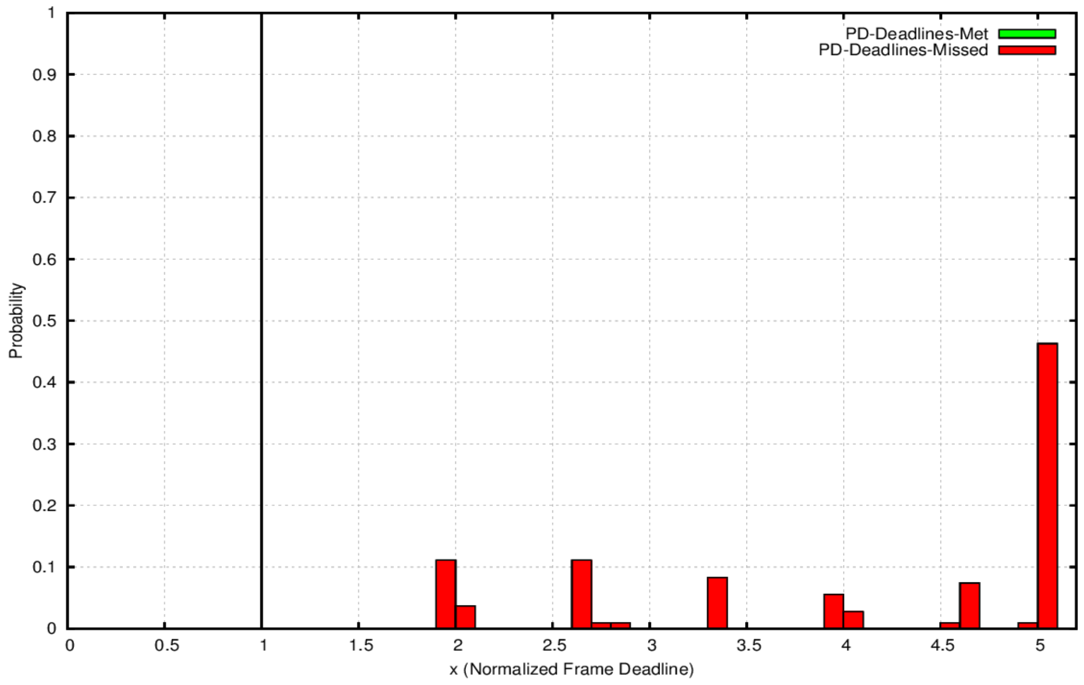


Figure 6.18) Application 3, PD-FFTs using FG-SDS (HPF) scheme

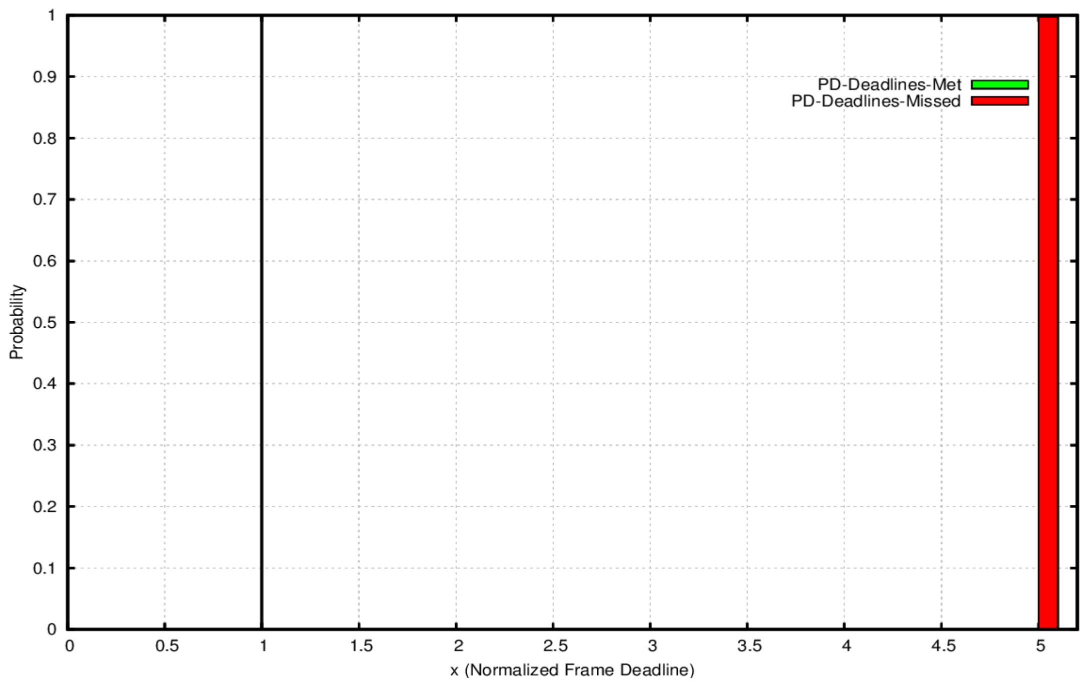


Figure 6.19) Application 4, PD-FFTs using FRRS scheme

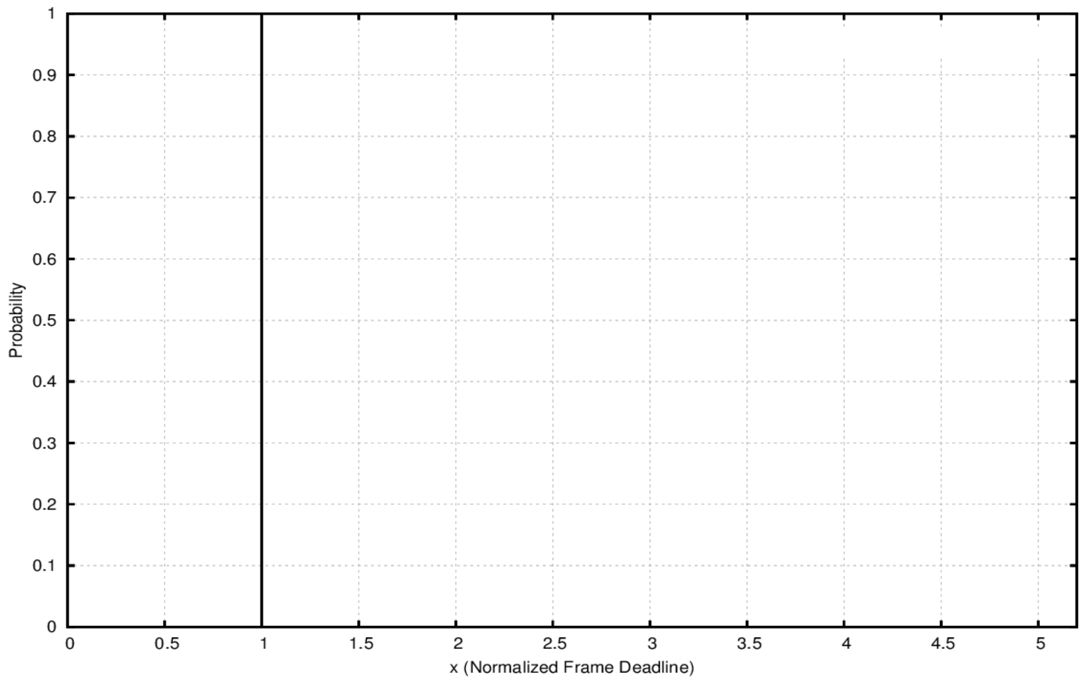


Figure 6.20) Application 4, PD-FFTs using FG-SDS (HPF) scheme

6.5. Summary

From the results of our evaluation we can safely say that there is very low probability that the highest priority application can miss its frame deadlines, if the schedule is feasible, using our FG-SDS model with HPF policy. However, since this model depends on the predicted execution time values of the command blocks for making scheduling decisions therefore the degree of accuracy in these values can change the behavior. To the best of our knowledge, to guarantee that none of the frames miss their deadlines may not be possible without preemption.

7. CONCLUSION AND FUTURE WORK

7.1. Conclusion

In this work we have seen that the GPU’s non-preemptive behavior is the main obstacle in the way of GPU multitasking. To overcome this challenge we proposed our FG-SDS model. By using this model applications can accumulate their commands into fine-grained command blocks before submitting them to the GPU. The execution time for each individual command block is also known by using the prediction in advance. The scheduler then uses these predicted execution time values of the command blocks and the priorities of the applications to select one of them to dispatch its command blocks to the GPU. This selection is based on HPF scheduling policy.

Using HPF policy the highest priority application always gets a chance to dispatch its command blocks whenever it becomes ready. The other low priority application can dispatch their command blocks only if the highest priority application is not ready and it has finished its current frame.

We used two different scenarios to evaluate the performance of FG-SDS model with HPF policy. The results suggest that this model has significantly improved the possibility for the highest priority application to meet its frame deadlines.

7.2. Future Work

Since the applications use CPU to dispatch their commands to the GPU, the CPU priorities of applications have to be set accordingly so that we have same priority order for the applications, both on the CPU and the GPU. Therefore one of the directions of this work is that we set the priorities in the same order and then evaluate the performance of the FG-SDS model.

Another direction is to quantify the scheduling overhead of the FG-SDS model with different CPU loads. By having this information the decision making of the scheduler can be improved. The scheduling algorithm may also be modified accordingly.

To further improve the performance we have to understand the GPU driver’s behavior in detail.

8. BIBLIOGRAPHY

1] S. Borkar, N. P. Jouppi and P. Stenstrom, "Microprocessors in the era of terascale integration," in *DATE '07*, 2007.

2] M. Hilbert and P. López, "The World’s Technological Capacity to Store, Communicate, and Compute," *Science*, vol. 332, pp. 60-65 , 2011.

3] P. Brucker, Scheduling algorithms, Springer, 2007.

4] G. C. Buttazzo, Hard real-time computing systems: predictable scheduling algorithms and applications, Springer, 2011.

5] "A Critical History of Computer Graphics and Animation," [Online]. Available: http://design.osu.edu/carlson/history/lessons.html.

6] [Online]. Available: http://www.b-u.ac.in/sde_book/compu_graphic.pdf.

7] Wikipedians, 3D Rendering, http://books.google.de/books?id=1GZiB-A5Y2kC&lpg=PR1&pg=PR1#v=onepage&q&f=false.

8] NVIDIA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.

9] A. Jenkins, "TopTenREVIEWS," TechMediaNetwork, [Online]. Available: http://computers.toptenreviews.com/gaming-laptops/graphics-cards-integrated-vs.-dedicated.html.

10] Daimler, [Online]. Available: http://www.daimler-technicity.de/en/f-125/.

11] A.-L. Dorofte, "Mercedes-Benz-Blog," Daimler, 26 10 2011. [Online]. Available: http://mercedes-benz-blog.blogspot.de/2011/10/telematics-future-of-mercedes-benz.html.

12] M. B. a. A. D. a. T.-c. Chiueh, "Graphic Engine Resource Management," in *Fifteenth Annual ACM/SPIE Multimedia Computing and Networking Conference (MMCN'08)*, 2008.

13] A. Dwarakinath, "A Fair-Share Scheduler for the Graphics Processing Unit," Stony Brook University, 2008.

14] K. Shinpei, L. Karthik, R. Ragunathan and I. Yutaka, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIXATC'11*, Portland, 2011.

15] J. Adriaens, K. Compton and N. Sung Kim, "The Case for GPGPU Spatial Multitasking," in *HPCA*, 2012.

16] SGI, "OpenGL - The Industry's Foundation for High Performance Graphics," [Online]. Available: http://www.opengl.org/.

17] Khronos Group, "OpenGL ES - The Standard for Embedded Accelerated 3D Graphics," [Online]. Available: http://www.khronos.org/opengles/.

18] "DRI Wiki," freedesktop, [Online]. Available: http://dri.freedesktop.org/wiki/.

19] M. Thielefeld, "Analyse und Evaluation der Ausführungszeit von OpenGL ES 2.0-Befehlen in Abhängigkeit von Parametern und Kontext," University of Stuttgart, Institute of Parallel and Distributed Systems, Stuttgart, 2012.

20] B. Paul, "Introduction to the direct rendering infrastructure," *Linux World,* 2000.

21] B. Paul and a. others, "The Mesa 3D Graphics Library," [Online]. Available: http://www.mesa3d.org/intro.html.

22] [Online]. Available: http://nouveau.freedesktop.org/wiki/.

23] Windows, [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff570589(v=vs.85).aspx.

24] "Windows," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/jj553428(v=vs.85).aspx.

25] Khronos Group, "Khronos OpenGL ES API Registry," 02 11 2010. [Online]. Available: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

26] Khronos Group, "Khronos EGL API Registry," 04 12 2006. [Online]. Available: http://www.khronos.org/registry/egl/specs/eglspec.1.3.pdf.

Khronos Group, "EGL - Native Platform Interface," [Online]. Available:

27] http://www.khronos.org/egl.

28] A. Munshi, D. Ginsburg and D. Shreiner, OpenGL ES 2.0 Programming Guide, Khoronos Group, Inc., 2009.

29] Khronos Group, "Khronos OpenGL ES API Registry," 12 05 2009. [Online]. Available: http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf.

30] Wikipedia, Polygon Mesh, [Online]. Available: http://en.wikipedia.org/wiki/Polygon_mesh.

31] Freescale. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q.

32] [Online]. Available: http://www.freescale.com/.

DECLARATION

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 05.07.2013