Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Electrónica
Programa de Maestría en Electrónica

Ray Tracing acceleration through a custom scheduling policy to take advantage of the cache affinity in a Linux-based Special-Purpose Operating System

Documento de tesis sometido a consideración para optar por el grado académico de Maestría en Ingeniería en Electrónica con Énfasis en Sistemas Embebidos

Alvaro Camacho Mora

Cartago, 30 de agosto de 2021

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Alvaro Camacho Mora

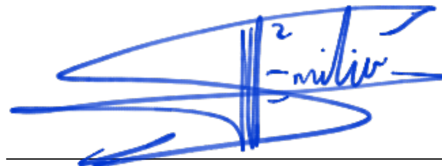
Cartago, 30 de agosto de 2021

Cédula: 3 0473 0491

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Tesis de Maestría
Tribunal Evaluador

Tesis de maestría defendida ante el presente Tribunal Evaluador como requisito para optar por el grado académico de Maestría en Ingeniería en Electrónica con Énfasis en Sistemas Embebidos, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal



Dr.-Ing. Juan Carlos Rojas Fernández Profesor Lector	M.Sc.-Ing. Marco Madrigal Solano Profesor Lector
Dr.-Ing. Jorge Castro-Godínez Profesor Lector	M.Sc.-Ing. Ernesto Rivera Alvarado Profesor Asesor

Los miembros de este Tribunal dan fe de que la presente tesis de maestría ha sido aprobada y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 30 de agosto de 2021

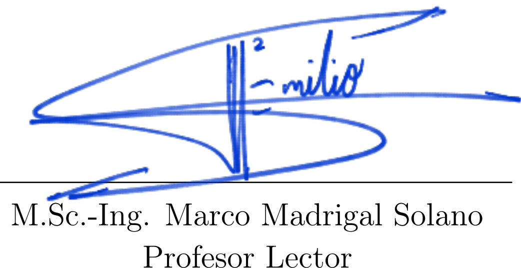
Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Tesis de Maestría
Tribunal Evaluador
Acta de Evaluación

Tesis de maestría defendida ante el presente Tribunal Evaluador como requisito para optar por el grado académico de Maestría en Ingeniería en Electrónica con Énfasis en Sistemas Embebidos, del Instituto Tecnológico de Costa Rica.

Estudiante: Alvaro Camacho Mora

Nombre del Proyecto: *Ray Tracing acceleration through a custom scheduling policy to take advantage of the cache affinity in a Linux-based Special-Purpose Operating System*

Miembros del Tribunal



Dr.-Ing. Juan Carlos Rojas Fernández
Profesor Lector

M.Sc.-Ing. Marco Madrigal Solano
Profesor Lector

Dr.-Ing. Jorge Castro-Godínez
Profesor Lector

M.Sc.-Ing. Ernesto Rivera Alvarado
Profesor Asesor

Los miembros de este Tribunal dan fe de que la presente tesis de maestría ha sido aprobada y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Nota final de la Tesis de Maestría: **95**

Cartago, 30 de agosto de 2021

Resumen

Esta investigación explora el beneficio de diseñar una política de calendarizacion personalizada que reduzca el tiempo de de ejecución de cargas computacionalmente intensivas. Cargas computacionalmente intensivas tales como *ray tracing*, son sensibles al cambio de contexto producido por el calendarizador. La política de calendarización propuesta asigna afinidad de cache fuerte para reducir el cambio de contexto al permitir que cada hilo tenga asignado un único núcleo para su ejecución. Utilizando un sistema operativo de propósito especifico, hipotéticamente, el sistema tendrá un mayor rendimiento al combinarlo con la política de calendarización personalizada. El algoritmo de ray tracing fue seleccionado como carga computacionalmente intensiva para comparar su rendimiento en un sistema operativo de propósito especifico contra un sistema operativo de propósito general con su configuración por defecto. Comparado a la referencia, ANOVA factorial confirmo un 19% de reducción en el tiempo de sintetizado promedio al usar la política de calendarización personalizada en un sistema operativo de propósito especifico.

Palabras clave: *ray tracing, sistema operativo de propósito especifico, SOPS, sistema operativo de propósito general, SOPG, afinidad de cache, afinidad de cache fuerte, afinidad de cache débil, ANOVA, cambio de contexto, cache miss.*

Abstract

The present research explores the benefit of designing a custom scheduling policy to reduce the execution time for computationally intensive workloads. Computationally intensive workloads, such as, ray tracing, are sensible to the context switching produced by the scheduler. The proposed custom scheduling policy assigns hard cache affinity to reduce the context switching by allowing each thread to use only one core during the process execution. Utilizing a special-purpose operating system will hypothetically boost the reduced execution time by integrating the custom scheduling policy. Ray tracing algorithm was selected as the computationally intensive workload to compare its performance in the special-purpose operating system with the custom scheduling policy against a general-purpose operating system with the default configuration. Compared to the baseline, the factorial ANOVA test confirmed an average 19% reduction of the rendering time using the custom scheduling policy in a special-purpose operating system.

Author Key-words: *Ray Tracing, Special-Purpose Operating System, SPOS, General -Purpose Operating System, GPOS, Cache Affinity, Hard Cache Affinity, Soft Cache affinity, ANOVA, context switching, cache miss.*

All We Need is Love..

Acknowledgments

I would like to thank professor Ernesto River, M.Sc. for all his valuable inputs during the development of this research. I also would like to thank my family for their support during this years of studies and career. To my father in the heaven. Additionally, I would like to thank Yos for her support and be an inspiration to finish this research. Finally, I would like to thank my friends who also helped to balance the load of working and studying.

Contents

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Problem . . . 6

1.2 Document Organization . . . 7

2 Theoretical Framework **9**

2.1 CPU . . . 9

2.1.1 Processor type . . . 10

2.1.2 Core count . . . 10

2.1.3 Clock Speed . . . 10

2.1.4 Cache Memory . . . 11

2.2 Operating System . . . 13

2.2.1 General-purpose Operating System . . . 14

2.2.2 Special-purpose Operating System . . . 15

2.2.3 Scheduler . . . 15

2.3 Ray Tracing . . . 19

2.4 Related work . . . 21

2.4.1 General-Purpose Kernel Optimization . . . 22

2.4.2 Completely Fair Scheduler Improvements . . . 23

2.4.3 Hardware Architecture Improvements . . . 24

2.4.4 Alternative methods to improve memory affinity . . . 25

3 Hypothesis and Objectives **27**

3.1 Hypothesis . . . 27

3.2 Objectives . . . 28

3.2.1 Main Objective . . . 28

3.2.2 Specific Objectives . . . 28

3.2.3 Deliverable . . . 28

3.3 Scope and Limitations . . . 29

4 Methodology **31**

4.1 Experiment Design . . . 31

4.2 Factors and Levels . . . 31
4.3 Measures and Combinations of the Experiment . . . 33
4.4 Response Variable . . . 33
4.5 Hardware Used for the Experiments . . . 34
4.6 Scenes Used for the Experiments . . . 35
4.7 Hard Cache Affinity used for the Experiments . . . 39
4.8 Rendering Library used for the Experiments . . . 40
4.9 Special-Purpose Operating System used for Experimentation . . . 40
4.10 Data Collection . . . 41
4.11 Factorial Analysis of Variance (ANOVA) . . . 41
4.11.1 Assumptions of the Factorial ANOVA . . . 42
4.11.2 Hypothesis for ANOVA . . . 42

5 Design 45
5.1 Yocto Implementation . . . 45
5.2 Strategy Selected for scheduling policy implementation . . . 46
5.2.1 Writing a new source code for the scheduling policy . . . 48
5.2.2 Source Code . . . 51
5.3 Adding the custom scheduling policy in the GPOS . . . 51
5.4 Limitations and Requirements . . . 51

6 Results 53
6.1 Cache optimization observed due to the custom scheduling policy . . . 53
6.2 Performance comparison of custom scheduling policy against other ANOVA factor combinations . . . 54
6.3 ANOVA Assumptions Tests . . . 58
6.4 ANOVA Results . . . 62
6.5 Obtained Metrics . . . 72
6.5.1 Specific scenario . . . 72
6.5.2 SPOS with scheduling policy comparison . . . 73
6.5.3 Actual rendering time reduction . . . 74
6.6 Correctness . . . 74
6.7 Output images . . . 75
6.8 Other results . . . 80

7 Discussion 83

8 Conclusions and Future Work 89
8.1 Conclusions . . . 89
8.2 Future Work . . . 90

Bibliography 91

A Implemented source code 99

B Gathered Data from Experiments **117**

C Compiling a custom kernel for Ubuntu OS **147**

D Recipe to include PBRT-v3 in the SPOS based on Yocto Project **151**

D.1 Yocto Recipe to Implement 151

D.1.1 Prerequisites to build PBRT from source code 152

D.1.2 Creating the Yocto's Recipe for PBRT-v3 153

List of Figures

2.1 Memory hierarchy in a computational system [1] . . . 12
2.2 Example of a Read-Black Tree [2] . . . 18
2.3 Ray tracing algorithm [3] . . . 20
2.4 Ray tracing image creation [4] . . . 21
2.5 A scene rendered using Rasterization (left) versus a scene rendered using ray tracing (right) [5] . . . 22

4.1 Scene for rendering: Barcelona Pavilion [6] . . . 35
4.2 Scene for rendering: Bathroom [6] . . . 36
4.3 Scene for rendering: Breakfast [6] . . . 36
4.4 Scene for rendering: Contemporary Bathroom [6] . . . 37
4.5 Scene for rendering: Crown [6] . . . 37
4.6 Scene for rendering: Landscape [6] . . . 38
4.7 Scene for rendering: Volume Caustic [6] . . . 38

5.1 Flowchart for the custom scheduling policy . . . 50

6.1 Mean plot for rendering time in function of the scheduling policy . . . 55
6.2 Mean plot for rendering time in function of the OS . . . 56
6.3 Mean plot for rendering time in function of the ANOVA's factors . . . 56
6.4 Mean plot for rendering time in function of the OS, scheduling policy, and scenes with maximum resolution . . . 57
6.5 Residual in function of the Normal Quantile . . . 59
6.6 Residual normality test for rendering time data. . . . 59
6.7 Residual by predicted plot . . . 60
6.8 Homogeneity of variance . . . 60
6.9 Box-Cox transformation result. . . . 61
6.10 Residual in function of the Normal Quantile with rendering time transformed 62
6.11 Residual normality test for transformed rendering time data. . . . 62
6.12 Residual by predicted plot with data transformed . . . 63
6.13 Homogeneity of variance with data transformed . . . 63
6.14 ANOVA effect test result . . . 64
6.15 Mean plot for Rendering Time (h) in function of the OS/scheduling policy 66
6.16 Mean plot for Rendering Time (h) in function of the OS/scheduling policy with sweep of image resolution . . . 66

6.17 Mean plot for Rendering Time (h) in function of the OS/scheduling policy with different scenes . . . 69
6.18 Mean plot for Rendering Time (h) in function of the OS/scheduling policy with a combination of different scenes and image resolution . . . 71
6.19 Correctness check using output images. . . . 74
6.20 Barcelona Pavilion scene at resolution of 1280x720 . . . 75
6.21 Bathroom scene at resolution of 1280x720 . . . 76
6.22 Breakfast scene at resolution of 1280x720 . . . 76
6.23 Contemporary Bathroom scene at resolution of 1280x720 . . . 77
6.24 Crown scene at resolution of 1280x720 . . . 78
6.25 Landscape scene at resolution of 1280x720 . . . 79
6.26 Volume Caustic scene at resolution of 1280x720 . . . 79
6.27 System running SPOS with PBRT-v3 in a x86 architecture . . . 80

C.1 Kernel configuration. . . . 148

List of Tables

4.1 Factors and levels of the experiment. 34
4.2 Hardware characteristics for the experiments. 34

6.1 Profiling results 54
6.2 Nomenclature for Tukey's HSD tables 65
6.3 Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP 65
6.4 Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with three resolution scenes 67
6.5 Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with seven scenes 68
6.6 Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with seven scenes at resolutions 1280x720 70
6.7 Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP (lower is better) 72
6.8 Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP at higher resolution (lower is better). 73
6.9 Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP at higher resolution and most complex scene (lower is better). 73
6.10 Performance improvement of the SPOS:CSP against SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP. 73
6.11 Actual rendering time reduction 74

B.1 Data obtained from the experiments 118

Listings

A.1 kconfig configuration option entry . . . 99
A.2 Scheduling policy macro definition . . . 100
A.3 Code inside custom.h header. . . . 100
A.4 Adding data struct to manage PBRT tasks . . . 101
A.5 Initializing custom_rq data structure . . . 101
A.6 Adding data struct to manage PBRT tasks . . . 102
A.7 Creating custom_task.h header. . . . 103
A.8 Changes at *struct task_struct* . . . 104
A.9 Adding custom scheduling policy source code . . . 105
A.10 Function to identify if the task is a PBRT task . . . 106
A.11 Function to set cache affinity in the new custom scheduling policy . . . 107
A.12 Function to add custom task (PBRT) to the run queue . . . 109
A.13 Function to remove custom task (PBRT) to the run queue . . . 111
A.14 Adding Custom Policy to the priority hierarchy . . . 113
A.15 Changing *setscheduling()* function . . . 114
D.1 PBRT-v3 Recipe for Yocto . . . 154

Chapter 1

Introduction

Time is money! This phrase reflects the reality of a globalized world where the demand for high-quality audiovisual content increases day by day [7]. In this context, the advancement of computer technologies plays an essential role in meeting the necessity of audiovisual content in a tight market calendar [8].

Rendering is converting scenes in 3-dimensions (3D) to build images in 2-dimensions (2D). This process takes so long because of the combination of textures, objects, and effects that the system must process to convert to a image that we can interpretate. The average duration of a movie or animation is 106 minutes at 24 frames-per-second (FPS), for a total of $24 \times 106 \times 60 = 152640$ images [9]. Two examples are world-renowned companies such as DreamWorks and Pixar, specializing in the creation of audiovisual content. The artists of those companies use rendering to create the scenes that we will watch in a movie, a short film, or a video game [10]. Rendering is a process that demands large amounts of computational resources, tangible when analyzing the rendering time of a single frame. For example, the rendering of each frame for the movie Toy Story 4 took around 60 to 160 hours [11]. Another example is knowing the size of Pixar's 'Render Farm', which consists of a supercomputer with 2000 machines and 24000 cores; even so, it took 2-years to render the entire Monsters University movie [12].

As mentioned in [13, 14], there are many rendering algorithms. However, one of the

most elegant and versatile algorithms is ray tracing, one of the most used audiovisual industries for offline rendering. Ray tracing offers superior photo-realism due to its intrinsic characteristics of reflection, refraction, and shadowing complexity [15]. When we use ray tracing as a rendering algorithm, the resulting image will be closer to how the human eye perceives the objects.

The ray tracing algorithm (explained in detail in the theoretical framework section) recreates the behavior of light through a relatively simple algorithm based on the emission of rays from the observer's camera to the object in question. With this, it is possible to recreate the interaction of light with the object [15]. However, it is a computationally intensive algorithm. Therefore, ray tracing has not been extended to real-time rendering (e.g., video game industry), where each frame must refresh the screen in a manner of milliseconds [16].

In recent years, companies such as NVIDIA have introduced the concept of real-time ray tracing. However, they offer a hybrid rendering process, combining rasterization (reconstruction of the image from geometric figures) and ray tracing (mainly for shadowing). [17, 18, 19].

There are many ways to execute a ray tracer. However, there is a drawback from a computational point of view. There must be coherence between the rays. Therefore, the GPU (Graphics Processing Unit) is not suitable for this task since the execution of the ray tracing algorithm does not obey a known and predictive data structure [18]. Besides, there is an unavoidable bottleneck in the computer architecture due to the data transfer among the CPU (Central Processing Unit) and the GPU, meaning that ray tracing in a GPU would be slower [17]. So, that is why many of the ray-tracing rendering engines use the CPU [15].

The rapid progress made in recent years in the semiconductor industry has allowed CPUs to evolve to be capable of processing algorithms that represent a very high workload for the system, such as ray tracing [20]. Companies such as Intel or AMD offer new CPUs with more cores, peripherals, cache memory, and sizes every year. That makes them ideal

for executing computationally intensive workloads [21].

A significant advance in memory architecture has evolved into a cache architecture known as Non-Uniform Memory Access (NUMA). In modern architectures, NUMA provides an architecture where each core has its local memory (cache L1 and cache L2) and a global memory (cache L3) for communication among cores. Thus, the cache memory will populate the data of a thread to exploit the cache affinity [22, 23, 24, 25]. Cache affinity, defined deeply in the theoretical framework, is the potential of having a lower cache miss rate and context switching. Cache affinity can be set as hard cache affinity when the system restricts the cores that a process can access during its CPU time.

However, all these advances at the hardware level could be insignificant without the software infrastructure capable of correctly and efficiently managing all the available resources. This software infrastructure is the Operating System (OS). According to [26], the OS has two main functions: 1) to serve as an extension of the architecture or virtual machine and 2) to serve as a resource manager. The second function is critical for the correct and optimal functioning of a computer system. Moreover, the resource management function is essential when the target OS executes computationally intensive workloads like ray tracing [27].

The OS manages the available hardware resources through process control algorithms. It is in charge of managing the processing of the tasks that the CPU must execute [1]. One of the most significant algorithms is the scheduler. Schedulers are generally classified depending on how critical it is to meet each process's deadline. For applications such as offline rendering, a soft time scheduler may be sufficient because a delay in the deadline would not imply catastrophic damage to the system [1, 26].

Linux's scheduler, known as Completely Fair Scheduler (CFS), is one of the widely used schedulers, released with version 2.6.23 of Linux kernel. Its philosophy is to "simulate an ideal CPU in the real world", that is, to offer a fair distribution of the CPU to each task in the execution queue [28]. In a multi-core architecture, CFS could have issues balancing workloads among the processors of the architecture [29] [30]. Furthermore, according to

[23], a scheduler should exploit cache affinity. However, to the best of our knowledge, hard cache affinity has not been extensively used to accelerate computationally intensive workloads. That is why we ran some small experiments and collected evidence that shows no hard cache affinity when ray tracing is executed (i.e., the execution threads of the ray tracing algorithm "jumps" within the cores of the architecture during profiler test when core assigned for execution changed more than 300 times). Moreover, when hard cache affinity is applied, the rendering time is lower as the core assigned for execution is the same during all the rendering time. The maximum rendering time reduction observed was up to 23%. As a result, we assume that assigning hard cache affinity for ray tracing execution can reduce rendering time.

A Linux-based general-purpose operating system (GPOS) should handle all the processes that the user demands, such as document creation, web browsing, and multimedia playback [1]. Likely on many occasions, due to the nature of the GPOS, the execution of tasks such as ray tracing can be temporarily suspended to attend other processes with higher priority, resulting in a potential increment in the execution time [31]. Furthermore, when the task most likely receives processor time, it will continue its execution in a different processor because CFS assigns soft cache affinity by default.

A special-purpose operating system (SPOS) potentially decreases the overhead caused by unnecessary tasks available in a GPOS (i.e., the SPOS will have only the essential tasks that ensure its stability) [32, 33, 34]. In addition, the cache misses can be reduced if a custom scheduling policy assigns hard cache affinity to the task. Thus, in a SPOS with a scheduling policy that assigns hard cache affinity, the execution time of ray tracing jobs may be reduced because the OS, potentially, has less overhead.

Therefore, the research reported in this document will focus on evaluating the potential increase of performance obtained by optimizing the execution of a highly demanding algorithm such as ray tracing through the design of a custom scheduling policy that explodes the benefits of assigning hard cache affinity. Moreover, to boost the rendering time reduction, a SPOS is build to integrate the custom scheduling policy. This implementa-

tion will be compared against the baseline operating system with the default scheduling policy.

1.1 Problem

As far as we know, there is no information on the potential use of a custom scheduling policy that assigns hard cache affinity in a special-purpose operating system to improve the performance of ray tracing algorithm. The evaluation will focus on three edges:

1. Comparing the execution time of ray tracing on a general-purpose OS with CFS and with a custom scheduling policy.
2. Comparing the ray tracing execution on a special-purpose OS with CFS and a custom scheduling policy.
3. Quantifying the performance gain between a general-purpose OS with CFS and a special-purpose OS with the custom scheduling policy.

Also, traditionally the execution of ray tracing for audiovisual development happens in high-end equipment [35, 36, 37]. However, there is a lack of information on optimizations in "budget-segment" systems to execute ray tracing which is the best-selling segment. Indeed, this research proves the potential to process ray tracing for real-time rendering, as is acclaimed in the video game industry.

The rendering time will be longer in a "budget-segment" system [21]. However, the experiment will use real-life scenes since one of the goals is to verify the potential improvement of ray tracing in the audiovisual industry. These scenes could be present in next-generation video games, movies, or short films.

1.2 Document Organization

This document presents the statistic evaluation of the feasibility of using hard cache affinity as a potential improvement point in executing computationally intensive tasks such as ray tracing. Also, the implementation of a special-purpose OS as possible performance improvement is analyzed.

This chapter presents a brief **Introduction** to the use of the ray tracing algorithm in the industry. Also, it describes the hardware/software elements that could influence the execution of the ray tracing algorithm.

Chapter 2 sets out the **Theoretical Framework** necessary to understand the development of this research. Besides, at the end of this chapter, a brief description of the related work on this topic is presented.

Chapter 3 exposes the **Hypothesis and Objectives** of this investigation. Along with this, the objectives and deliverables are raised and described. The scope and limitations of this investigation are also detailed.

Chapter 4 presents the **Methodology** proposed for this research. We describe the software and hardware infrastructure used for the evaluation of this research. Also, the design of the experiment and the statistical method followed for the data analysis.

Chapter 5 presents the **Design** and the strategy to accelerate computationally intensive workloads like ray tracing through the cache affinity in a SPOS.

Chapter 6 displays the **Results** obtained through the experiment.

Chapter 7 presents the **Discussion** of the results obtained after results analysis.

Finally, chapter 8 shows the **Conclusions and Future Work** resulted from this research effort.

Chapter 2

Theoretical Framework

In this chapter, we present the main theoretical concepts to have a better understanding of this research. Specifically, we describe the Central Processing Unit, Operating Systems, and ray tracing algorithm. Also, at the end of this chapter, we explain some related work to the scope of this research.

2.1 CPU

Nowadays, almost every electronic device uses a CPU as an electronic brain. This electronic brain, in addition to processing data, controls all the other components in the system [38]. Rather than early design, modern CPUs follow a decentralized approach, meaning that we may see more than one processor dedicated to controlling subsystems in the entire system. However, the main CPU is the most critical hardware component as it always controls and coordinates the overall system by telling the other processors when to start and stop [38, 39]. The CPU performance can be measured by analyzing four main parameters: core-count, cache size, processor type, and clock speed.

2.1.1 Processor type

According to [39], the CPU can be divided into two broader categories: CISC (complex instruction set computer) and RISC (reduced instruction set computer). This classification depends on the instruction set of the architecture. When we refer to CISC, it means an instruction set that usually includes many instructions (typically hundreds). Each instruction can perform any arbitrarily complex computation (e.g., one instruction manipulates graphics in memory and others compute the sine and cosine functions) [39]. In this research, we will focus on CISC because the architecture x86 uses this instruction set, and it is the most famous architecture by general-purpose computing [40].

2.1.2 Core count

In this context, core means a processor into the CPU die. Each core has its execution unit, control unit, and local memory. Modern CPU chips follow a multi-core architecture. There is an open 'battle' between CPU manufacturers to add more cores into their chips [41]. Multi-core architecture allows parallel execution, ideal for high-performance tasks because a single core cannot be clocked at arbitrarily high speeds. Multi-core architecture does not mean that doubling the core count will double the speed, as Amdahl's Law explains [1]. Amdahl's Law says that the performance will not rise in the same proportion as the increase in CPU cores because it is limited by the ratio of software processing that must be executed sequentially [42]. Communication among the cores is a factor that may reduce performance in this kind of architecture [17, 39].

2.1.3 Clock Speed

The clock speed indicates how fast the CPU can run [43]. This parameter is one of the most meaningful to understand the performance because, at higher clock frequency/speed, the CPU can execute more instructions in less time. Modern architectures bring two clock frequencies/speeds: the baseline (i.e., base frequency) and turbo frequency (i.e., the CPU

uses turbo frequency when increases the workload) [44]. However, the CPU cannot perform at turbo frequency for a long time due to thermal implications. This phenomenon is explained by Equation 2.1, where the average power calculation is performed by multiplying the switching factor (α), the capacitance (C), operation voltage (V_{dd}), and clock speed (F_{clock}) [39]..

$$P_{avg} = \alpha C V_{dd}^2 F_{clock} \tag{2.1}$$

Equation 2.1 demonstrates the influence of the clock speed in power consumption and heat increase. That is why the overclockers (people who try to increase the clock speed out-of-base values) need an advanced thermal solution to mitigate the thermal risks [45].

Another important fact is that multi-core architecture was born because of the increase in thermal issues derived from clock speed [1]. In the early 2000s, power consumption became a problem because, in single-core architecture, the clock speed increased from generation to generation. The designers realized that having a multi-core architecture at lower clock speeds may have the same computationally capability as a single-core with higher clock speed [1].

2.1.4 Cache Memory

Memory access is one of the main bottlenecks in modern architecture because of the latency introduced by retrieving data from the main memory [39]. The cache is a high-speed memory located between the processor and main memory, allowing the processor to have 95% of the data required by a program thanks to the principle of *locality* and its low latency [17].

Lower the cache level, faster and smaller (capacity). The majority of modern computational systems have at least three cache levels. Cache L1 and Cache L2 are assigned per core in a NUMA architecture, and Cache L3 is shared among the architecture's cores

[39].

Physically located at the same die as the cores, the cache has become one of the most significant components in a computational system, as shown in Figure 2.1. Access time determines the performance of the cache memory [39]. If data required by a core is not in the cache memory, every step down in the memory hierarchy will introduce latency to the system. This phenomenon is called *cache miss* in contrast to a *cache hit* that happens when the data is found in the cache. Design elements like replacement policy and associative will impact the performance by reducing (or increasing) the cache misses [25].

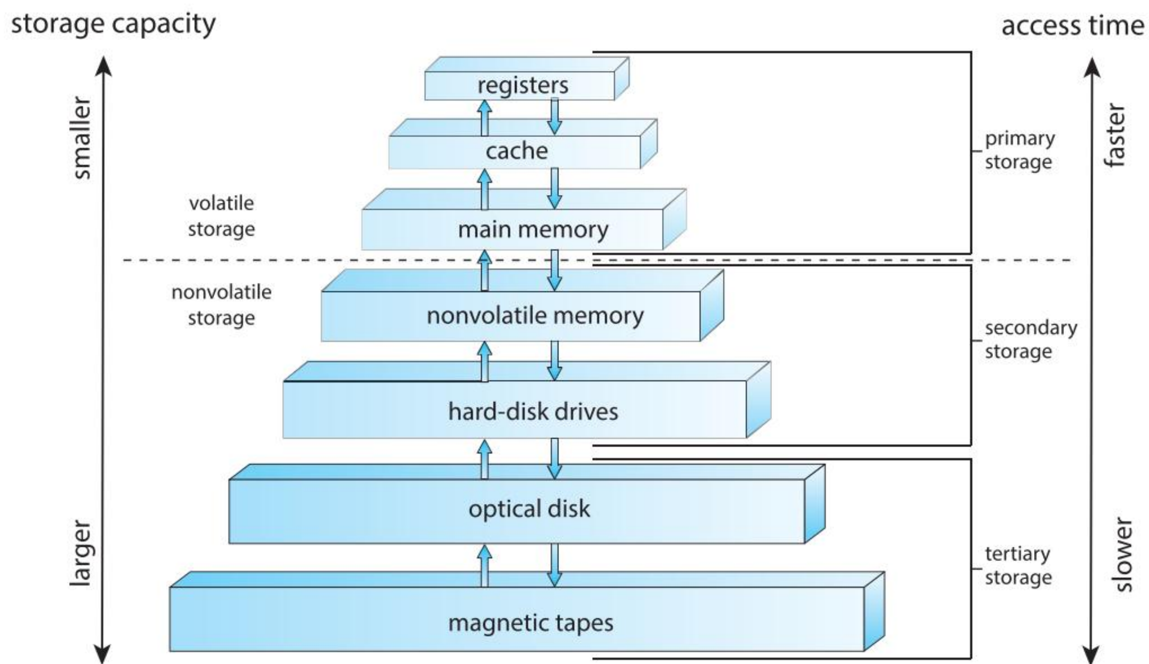


Figure 2.1: Memory hierarchy in a computational system [1]

Cache Affinity

As exposed above, a cache hit is a crucial element to ensure cache performance [1]. Consider what happens when a thread has been running in a specific processor (P1). The thread would populate the data most recently used in the cache. Likely because of a locality principle next time the thread asks for data, the data will be in the cache in a

mechanism known as "warm cache" (i.e., numerous cache hits). However, what happens if the scheduler, because of load-balancing, decides to move the thread from P1 to another processor (P2)? The cache in P1 will be invalidated (i.e., context switching [1]), and the cache in P2 must populate the data to run the thread, implying a massive cost for P2. The scenario described is what *cache affinity* wants to prevent by restricting the core(s) where a process/thread can run to have a warm cache [1, 22].

Linux Scheduler (Completely Fair Scheduler) allows two main cache affinity types: *soft cache affinity* and *hard cache affinity* [1]. On the one hand, soft cache affinity is the default condition because the scheduler has the policy to keep running a thread in the same processor, but not guaranteed because it can change depending on workload conditions. Furthermore, on the other hand, we have hard cache affinity when the user, through a system call, defines the core(s) where a thread is allowed to run. The Linux OS has a system call to assign hard cache affinity, *sched_setaffinity()*, but it can be inefficient and may create race conditions and memory allocation problems during the task's execution [46].

2.2 Operating System

The operating system (OS) transforms the computer from a useless lump of metal into a sophisticated electronic device [26] An OS allows us to do basic tasks like checking our social networks or playing music and videos and doing complex processes like controlling a rocket's launch or discovering treatments for disease. The OS performs fundamentally two functions: 1) extending the machine and 2) resource manager [1, 26].

[1] Context switching happens when a new task gets CPU time, and all the data in the cache must be renewed by retrieving data from the main memory, implying an enormous latency for the system [26].

Extending the machine

The OS provides the programmer a high-level abstraction of the hardware functionality (i.e., in most cases, the hardware is transparent to the programmer) [26].

Resource Manager

An easy example to understand the resource manager function of the OS is to imagine a system where three programs are ready to be executed. Each program wants to use the same resources. It may imply potential chaos in the system, but the OS will bring order to the system by knowing the availability of the computational resources. Also, the OS will prioritize each program to recognize when the computational resources should be assigned. Besides, it will protect the data of each program [26].

As a result of Moore's Law, computers are present everywhere; within our watch, refrigerator, cell phone, and TV [47]. As explained early, those electronic devices need an OS to have a correct operation. That is why the OS should adapt to the necessity of the system (e.g., the OS used by our personal computer is different from the one used by a toaster). For that reason, exists an OS classification: general-purpose OS and special-purpose OS [48].

2.2.1 General-purpose Operating System

The main reason for a general-purpose OS is to make a single OS that offers various services that work for a range of computers. Computer manufacturers follow this approach because it makes cheaper and scalable systems [26].

The objectives for such systems are to accommodate an environment of diverse applications and operating modes, leaving in second place objectives like increased throughput, lower response time, and adaptability [48].

Examples of a general-purpose OS are:

- Windows
- Mac OS
- Some Linux distributions like Ubuntu and CentOS.

2.2.2 Special-purpose Operating System

In contrast to a general-purpose OS, a special-purpose OS focuses on an optimized system to execute a task or a group of functions as efficiently as possible, that is, designed according to performance specifications [48]. There are special-purpose OSs such as μkernel or Exokernel implemented to fit the hardware. For that reason, a special-purpose OS could be more expensive than a general-purpose and lost some flexibility because it could be hardware-dependent [33, 34].

A classic example of a special-purpose OS is an RTOS (Real-time OS). In these systems, losing the deadline will have catastrophic consequences. As a result, an RTOS has precise resource management to meet the deadline and be predictable [49, 50].

2.2.3 Scheduler

Modern Operating Systems can get the most out of the hardware where they are running [1]. The OS does this by making each process "believe" that only it is being executed in the processor by giving the process the feeling that it is alone. The scheduler is an algorithm that assigns the processor time to each task in a run queue [51].

The scheduling idea is relatively simple [1]. The scheduler should send a process to run, wait until it finishes its execution, spend its processor time, and then send another task for execution. The main objective is to avoid having the CPU idle by having a queue with programs ready to run. When a program should wait, another process can take over the use of the CPU.

As scheduling criteria, numerous scheduling algorithms follow CPU utilization,

throughput, turnaround time, waiting time, and response time [26]. In general, we would like to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. However, the balance desired would depend on the target processes.

Many scheduling algorithms fulfill different necessities [1]. Some of the scheduling policies are listed below:

- Completely Fair Scheduler (CFS): Default scheduling policy for Linux-based OS (more details in next section).
- First-Come, First-Served (FCFS) Scheduling: This scheduling policy executes its run queued requests and processes in order of their arrival. It is considered the most straightforward CPU scheduling policy.
- Shortest-Job-First Scheduling: This scheduling algorithm choose the next task to be executed in the function of its execution time. The task with the shortest execution time is executed next. It has the advantage of reducing the average waiting time for process awaiting processor time.
- Round-Robin Scheduling: This algorithm is a preemptive scheduling policy. Each task will receive a fixed time to execute, which is called *quantum*. Once a task is processed for a given quantum, it is preempted, and another task executes for a given time.
- Priority Scheduling: It is a non-preemptive algorithm where each process receives a priority. A process with the highest priority is executed first, and so on. When two processes have the same priority, FCFS is applied.
- Multilevel Queue Scheduling: It uses other scheduling policies to group tasks with similar characteristics and the best scheduling policy for a given group of tasks.
- Multilevel Feedback Queue Scheduling: Similar to Multilevel Queue Scheduling. The main difference is that it allows a process to move between queues. The inten-

tion of moving tasks is that a task that uses too much CPU time can be moved to a lower priority queue, or a task that waits too long can be moved to a higher priority queue.

- Earliest deadline first scheduling (EDF): EDF is an optimal dynamic priority scheduling algorithm used in real-time systems. It assigns priorities to the task according to the absolute deadline. So, the task whose deadline is closest gets the highest priority.

- Rate Monotonic: It is a preemptive algorithm that belongs to the static priority scheduling category. As the EDF, it is mainly used in real-time systems. The priority of each task is decided according to the cycle time of the process that is involved. For example, if a process has a small job duration, it has the highest priority.

Completely Fair Schedule (CFS)

Scheduling is a fundamental operating-system function [1]. Talking about Linux OS, CFS is part of the Linux kernel since version 2.6.23 [52]. CFS is a scheduler that models an ideal, accurate, multi-tasking CPU on real-wrold hardware [28]. It means that CFS tries to model a CPU that can run several tasks in parallel, offering each of them the same processing power. An example of a simple process processor would receive 100% of the processor's power. Assuming a single-thread processor, if we have 2-processes, each one would have 50% of the processor's power. However, this fairness does not exist in a real processor because only one task can run on a processor and receive 100% of the processing power as long as it is assigned [52].

The central part of the CFS implementation is the run queue [51]. Rather than the traditional approach of a FIFO (first-in, first-out), CFS uses a red-black tree (RBTREE). The main characteristic of this implementation is the easier insertions and removals within the tree than other data structures [28]. Another advantage of the RBTREE, it is con-

sidered as an $O(log(N))$ problem (where N is the number of nodes in the tree), reducing the computational workload when going through the tree. Another feature is that this type of data structure is excellent for a hierarchical organization, as the virtual runtime requires (*vruntime*) [28].

Figure 2.2 shows the implementation of an RBTREE, where tasks are *sched_entity* objects in the RBTREE. On the left side of the tree, tasks with the highest CPU need (lowest vruntime) are stored, and on the right side of the tree, tasks with the lower CPU requirements (higher vruntimes) are stored. The scheduler, to be fair, chooses the leftmost node of the RBTREE as the next task to be executed (to maintain fairness) [52]. The tree's content migrates from the right to the left to keep it fair when a task is removed from the tree. Therefore, each executable task chases the other to maintain a balance of execution in the set of tasks. An expropriated task is sent back to the tree to the rightmost side to execute the new leftmost task [46].

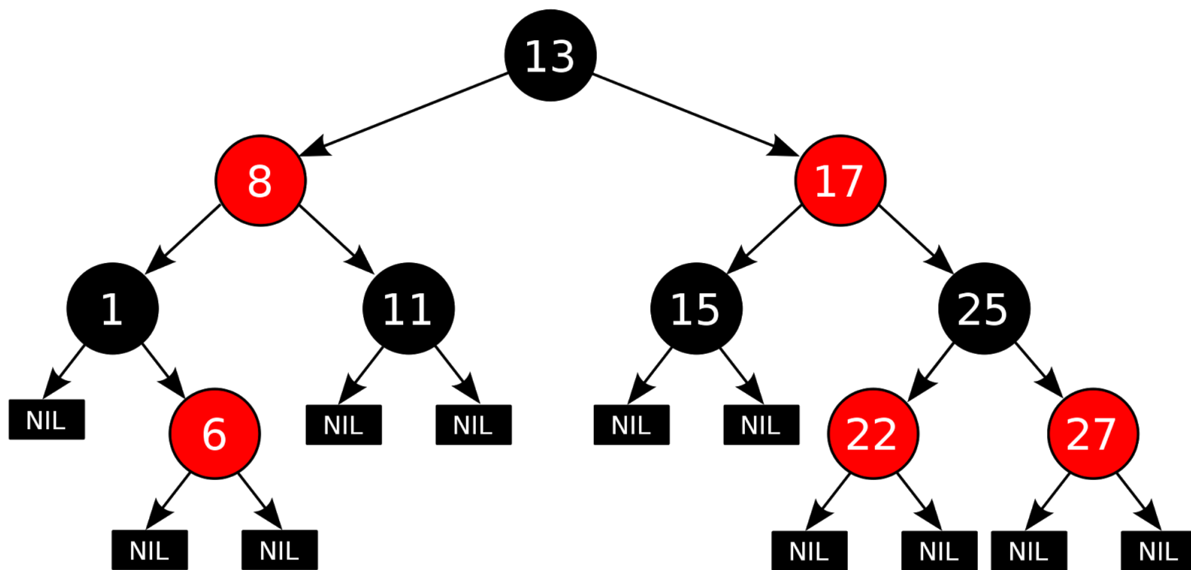


Figure 2.2: Example of a Read-Black Tree [2]

CFS must care about workload balancing and cache-affinity. By default, CFS has a soft cache affinity because it should follow workload balancing and cache-affinity principles. However, according to [46], there are mechanisms to force CFS to have a hard cache affinity, which is the primary purpose of this research effort. Nevertheless, [46] says that

forcing CFS to set hard cache affinity through a system call can be inefficient and may create race conditions and memory allocation problems during the task's execution.

2.3 Ray Tracing

Rendering is the method to transform 3D scenes into 2D scenes that we can watch on television, computer, and cinema. The traditional rendering technique, named rasterization, uses geometry figures to approximate the objects' shape, light, and shadow in a 3D scene [53]. This approximation is very computationally efficient, allowing real-time rendering in some popular applications like video games. On the other hand, almost all photorealistic rendering systems use the ray tracing algorithm because they are the most elegant and versatile [15].

In 1968, Arthur Appel was the first person who documented ray tracing for image synthesis [54]. Figure 2.3 shows that a mathematical ray originates from the origin point or 'eye point' crossing the image plane at the center of a pixel. For each ray, the system calculates which object in the scene intersects the ray. It also calculates which object is closest to the ray origin, which will be the visible object for the pixel. The Eye Point and the Image plane exist in the three-dimensional space [17]. When a ray does not intersect any object in the scene, the default color is assigned. However, if the ray intersects an object, the pixel will receive the color generated by the ray [3].

As [17] explains, the ray tracing algorithm can be described as a nested *for* in which the external *for* iterates through the y-axis and internal *for* iterates through the x-axis, resulting in intense use of the CPU. That is a simple abstraction of the ray tracing algorithm, but it helps to have a high-level understanding of the algorithm. There are formal and powerful rendering engines using ray tracing [55]. Some of these rendering engines are:

– Mitsuba

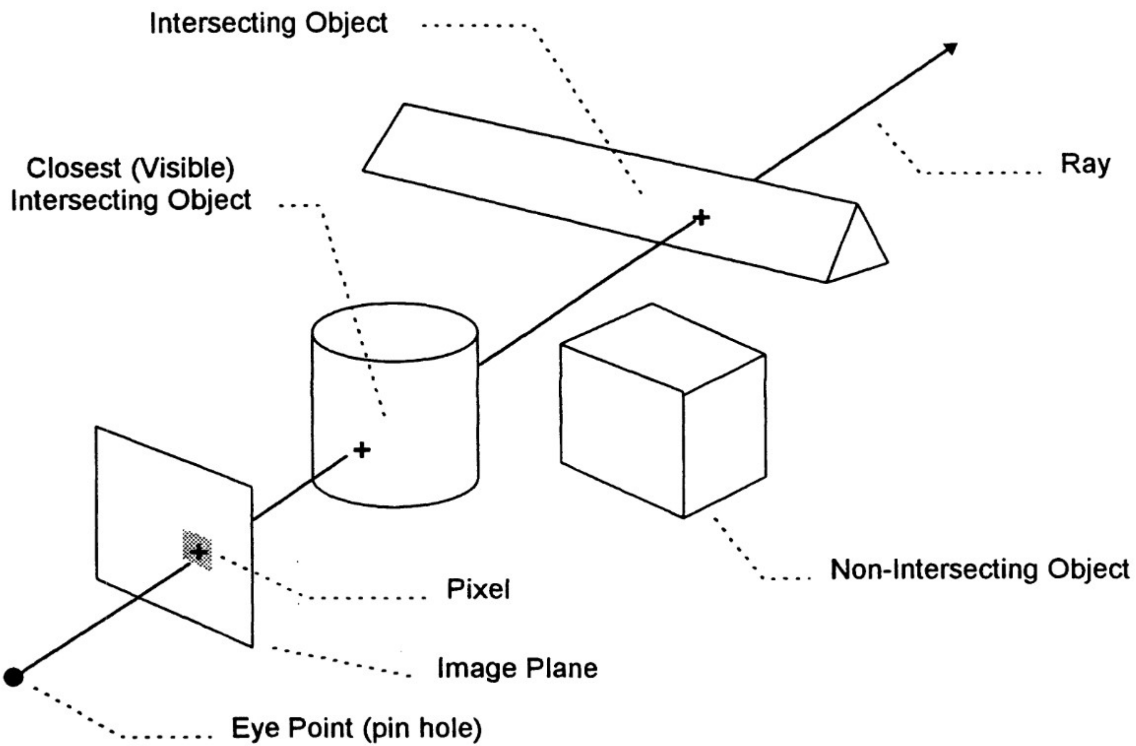


Figure 2.3: Ray tracing algorithm [3]

- PBRT-v3
- LuxRender
- Hyperion
- RenderMan

Figure 2.4 shows three relevant effects in a scene: color, illumination, and shadowing. The color and illumination depend on the intensity and position of the light source [55]. It is required to have at least one light source. Lambert's Law is the mechanism that generates the color shown in a rendered image by scaling on a range from zero to one of each one of the color components [17]. The shadowing happens when there is an obstacle between the source light and the object [15]. The easiest way to determine if a shadow exists is by creating a new ray called *shadow rays*. Their origin will be the object's surface, and its direction will be towards the light source. If there is no obstacle between the object and the light's source, the light's source contribution is included in the final image [15].

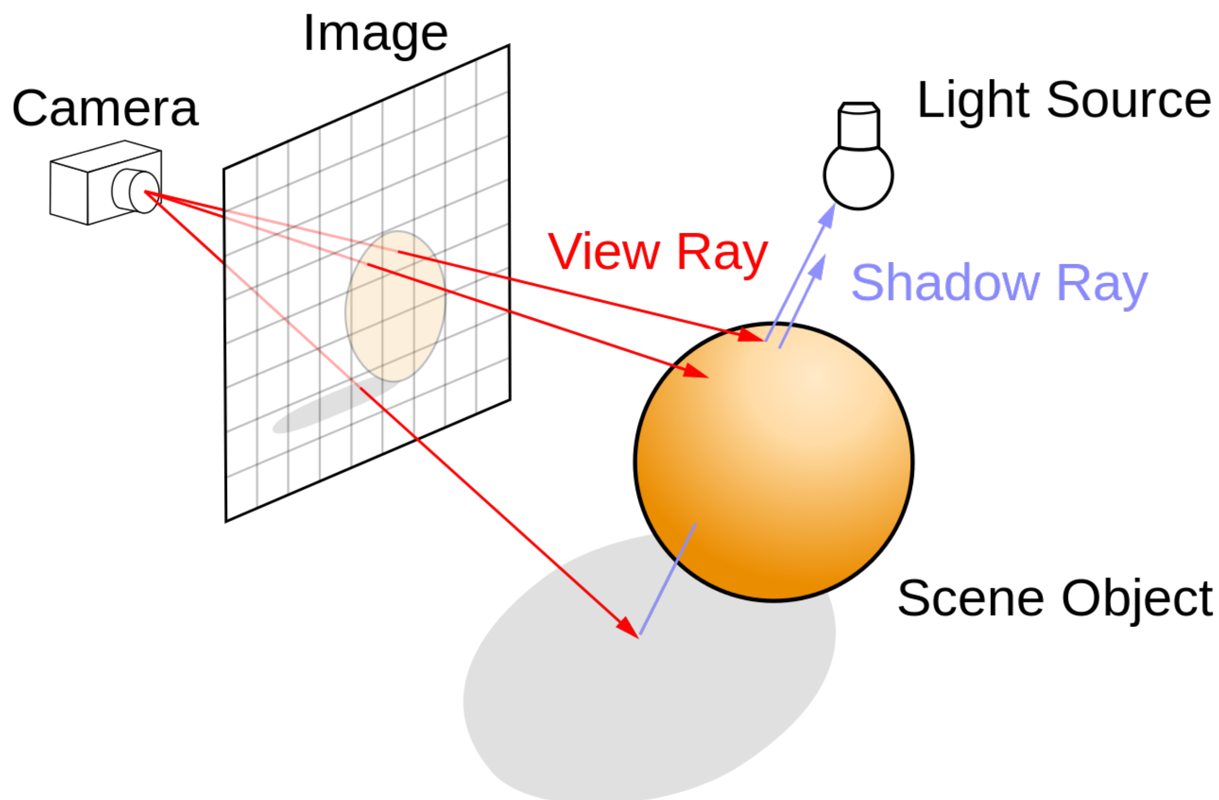


Figure 2.4: Ray tracing image creation [4]

Figure 2.5 shows the power of the ray tracing algorithm. The scene to the left uses rasterization, the traditional approach that follows in the video-game industry [18]. The enormous difference in the final scene is clear when ray tracing is used (image to the right). The reflection, illumination, and shadowing of the fire on the car are close to what we observe in real-life events.

2.4 Related work

This section presents different implementations found in the literature to accelerate tasks computationally intensive. The related work section has three areas: General-Purpose Kernel Optimization, Completely Fair Scheduler Improvements, and Hardware Architecture Improvements.

Figure 2.5: A scene rendered using Rasterization (left) versus a scene rendered using ray tracing (right) [5]

2.4.1 General-Purpose Kernel Optimization

An implementation of a reduced kernel can be called a μ-kernel. A μ-kernel is a reduced kernel that contains the essential components following two principles: independence and integrity. Besides, the μ-kernel is intended to allow flexibility and high performance.

As [33] analyzes, there are scenarios where a μ-kernel can be a perfect solution to improve system performance. It is mandatory for an excellent reduced kernel implementation that must include only the components needed to work correctly. The way to reach this is:

- **Optimize Address Spaces**: By optimizing how the operating system associates the physical page to the virtual page. This must be done so that the address space keeps hidden from the hardware to ensure security.

- **Threads and Process Scheduling**: Threads are executed within the address space, with characteristics of *register allocation, instruction pointer, stack pointer, and status information.* How it is associated directly with the address space should be implemented within the μ-kernel.

- **Unique Identifiers**: The unique identification of processes, threads, and tasks is required to establish efficient and reliable communication channels, so the μ-kernel must ensure this service. Other methods as cryptography can be used, but for local communication, it is a costly process.

The flexibility in a μ-kernel is met by having good memory management, correct multimedia resource allocation, and device driver. The main disadvantage of a μ-kernel is that the design entirely depends on the hardware (i.e., unique solution for each piece of hardware) [56].

2.4.2 Completely Fair Scheduler Improvements

As [51] and [46], the approach followed during Linux Scheduler design makes it "easy" for the developers to develop their scheduler and add it as a new scheduling policy, co-existing with the Completely Fair Scheduler (CFS). This section will be divided into scheduling strategies intended to boost the system's performance by optimizing Cache and Memory access.

One of the main problems of modern architectures is a large number of tasks/threads that are running in parallel [57]. They must share hardware resources such as cache, memory, CPU, and I/Os. It can cause resource contention, which will affect the performance of the threads. If the system knows that this containment will occur, the effect can be mitigated and compliance with performance, power consumption, and justice. Therefore, an algorithm that meets this criterion must have the following objectives [58, 59]:

1. To detect shared memory contention.

2. To depend solely on the information gathered by the monitoring mechanism in modern processors.

3. Not to require additional hardware.

Memory Link and Cache-Aware co-calendaring for architectures Chip-Level Multiprocessing (CMP). It is based on a classification scheme that monitors the use of resources throughout the entire memory hierarchy: from main memory to CPU cores. With this, it is possible to predict the interference that may occur between applications. Also, support a co-scheduling algorithm that exceeds the policies of standard scheduling in terms of performance and fairness of CPU usage [58].

As [29] demonstrates, CFS is not ideal for multi-core implementations because it fails to distribute the core workload efficiently. It happens because CFS is not scalable and fails in two functions: *load balancer* and *per-core fair-share scheduler*. The workload is distributed more efficiently between the more heavily loaded cores and the more lightly loaded ones by modifying those components [30].

CFS is intended to be a scheduler for general-purpose operating systems [51]. It can be modified and optimized to incorporate a more robust implementation by adjusting parameters like *scheduler_latency* and others; the execution time can be reduced [60]. Modifying the parameters of baseline CFS according to the necessity of the applications, there are performance improvements up to 10% [31].

Ray tracing relies on how well designed the scheduling at the cache level is [1, 15]. The threads being executed by a rendering algorithm 'jump' from one core to another to optimize CPU usage. Still, it may increase the cache misses by reducing the cache affinity [53]. There are some techniques like CFS+ that obtained up to 4.56% improvement in system throughput for the applications studied by detecting the cached content to avoid it or even correct it [61].

2.4.3 Hardware Architecture Improvements

Computationally intense tasks, like ray tracing, can be accelerated by modifying the hardware architecture, resulting in execution time reduction of algorithms like ray tracing. Examples of these architectures are Symmetric Multi-Processing (SMP), Non-Uniform

Memory Access (NUMA), and Simultaneous Multi-Threading (SMT) [57]. A similar idea is presented by [5], where it shows how NVIDIA created a specific GPU to run algorithms like ray tracing.

By having a hybrid optimization between software and hardware, the execution time of a computationally intense task can be reduced [62]. If optimization is done only considering the software side, the result is poor performance. On the other hand, if optimization is done only at the hardware level, it may reduce the execution time, but the complexity of the hardware grows. By having a hybrid implementation where both software and hardware are optimized, the improvement can be rounded to 42%.

[17] and [16] present a evaluation of the ray-tracing acceleration in low-cost hardware. An APU (Accelerated Processing Unit) is hardware with a CPU and a GPU integrated into the same die. Ray tracing was the computationally intensive workload chosen to compare its performance in an APU against the CPU and GPU. The main advantage of the approach followed is that it takes advantage of the ability of the APU to share and coordinate data within its internal processor. The author demonstrates a performance improvement of around 65% for a general case. For the most complex scenario, the author proves an increase in the performance of 79% against GPU and 51% against CPU.

2.4.4 Alternative methods to improve memory affinity

Modern systems have deep and complex memory hierarchies with multiple cache levels and memory controllers within a NUMA architecture. For such systems, mapping threads that share data cores with shared cache, cache usage, and mapping pages to memory controllers to reduce the access overload is required from [24, 63] perspective. kMAF is a mechanism that performs integrated thread and data mapping in the kernel by using the page faults of parallel applications to characterize their memory access and cache performance. The results demonstrate a maximum improvement in execution time reduction up to 35.7% and 34.6% in the energy efficiency. The authors point out that the

improvement observed in their implementation is mainly because of the memory affinity of parallel applications through the optimized thread and data mapping.

Another author, such as [64], uses cache affinity optimization to obtain up to 21% average speedup over the baseline execution. This improvement was achieved by developing a new dynamic concurrency controller for *TinySTM* (Software transnational memory). It also features an affinity-aware thread migration technique that fine-tunes thread placement.

The implementation exposed by [61] and [64] are essential papers that work as a baseline for this research effort. They show that the optimization in the cache affinity can improve the system's performance. The custom scheduling policy can also be considered an affinity-aware implementation. It pursues the hard cache affinity to improve computationally intensive workloads like the ray tracing algorithm.

Chapter 3

Hypothesis and Objectives

This chapter exposes the Hypothesis and Objectives of this investigation. Along with this, the objectives and deliverables are raised and described. The scope and limitations of this investigation are also detailed.

3.1 Hypothesis

In combination with a special-purpose OS, exploiting the cache affinity shows the potential to improve the performance of computationally intensive workloads like ray tracing. As a result, the hypothesis for this thesis reads as follows:

“Cache misses due to context switching is an issue that increases the execution time in processes computationally intensive such as ray tracing. Suppose the number of cache misses decreases through a custom scheduling policy that assigns hard cache affinity in a special-purpose OS. In that case, the execution time can be reduced by at least 10%[1] when compared against to a general-purpose OS using default CFS scheduling policy."

[1] 6.3%, 6.8%, and 3.58% was the average performance improvement between 2012 and 2019 in CPU processors (for multi-thread workloads) for laptop, desktop, and server; respectively [65].

3.2 Objectives

3.2.1 Main Objective

Evaluate the Ray Tracing acceleration through a custom scheduling policy that assigns hard cache affinity optimization in a special-purpose operating system against a general-purpose operating system using default.

3.2.2 Specific Objectives

1. Identify the state-of-art design of special-purpose operating systems and optimize through cache affinity for computationally intensive workloads algorithms like ray tracing.
2. Design a solution that integrates a custom scheduling policy that assigns hard cache affinity in a special-purpose operating system.
3. Validate the solution designed through a design of experiments.

3.2.3 Deliverable

This section summarized the deliverables proposed for each of the specific objectives defined in the previous section.

Specific Object 1: It documents the state-of-art and related work on designing special-purpose operating systems and optimizing cache affinity for computationally intensive workloads algorithms like ray tracing.

Specific Object 2: Present the system designed that includes a custom scheduling policy that assigns hard cache affinity in a special-purpose operating system.

Specific Object 3: Provide a summary of the statistical data obtained from the experiments.

3.3 Scope and Limitations

The technology field is an ever-changing environment. That is why this research has taken numerous assumptions to clarify and delimit thesis scope and results. Any consideration outside of what is being defined in the objectives, deliverables, and this section is considered outreach of this thesis.

Hardware used: By the time of this research, a high-end gaming laptop can cost up to $6499, but the average cost for a high-end gaming laptop range between $2000 and $3000 [66]. Because of hardware availability and to limit the scope of the results obtained, the "budget segment" for gaming laptops has been selected. The range of this segment is around $1000 [67]. The actual hardware used to benchmark the ray tracing algorithm is describing in the methodology section.

Selected library to render ray-tracing on the Linux environment: *PBRT-v3* was chosen. The reasons for using PBRT-v3 are discussed in chapter 4. Any other state-of-the-art library for ray tracing execution is regarded as future work.

Selected method to assign a hard cache affinity in the ray-tracing execution: Adding a new *scheduling policy* into the Linux kernel scheduler was chosen to set hard cache affinity. The reasons for using this approach are discussed in chapter 4. The use of any other procedure to assign hard cache affinity is regarded as future work.

Selected environment to build the special-purpose operating system: The *Yocto Project* was chosen. The reasons for using the Yocto Project are discussed in chapter 4. Using any other methodology/platform to build a special-purpose operating system is regarded as future work.

Scenes for rendering: The scenes used for the experiments are the examples provided by the designers of the PBRT-v3 library [6]. The experiments will consist in three resolutions: 640x360 (nHD), 960x540 (qHD), and 1280x720 (HD). Seven

real-life scenes will be part of the experimentation (more details of the scenes in chapter 4). Any other arrangement not contemplated in this research is regarded as future work.

Better performing hardware: The hardware selected for the experimentation was considered adequate to fits the necessities of this research effort. The use of newer or powerful hardware is regarded as future work.

Chapter 4

Methodology

This chapter presents the methodology followed in this research effort. First, a brief introduction to ANOVA and the details of the assumptions taken in this thesis.

4.1 Experiment Design

This thesis will evaluate several scenarios to understand the factors that affect the rendering time of ray tracing using the PBRT framework. A factorial ANOVA is an Analysis of Variance test with more than one independent variable (IV). Factorial ANOVA is an efficient way of conducting a test because instead of doing a series of experiments testing only one IV, ANOVA permits an assessment of all IVs simultaneously [68]. Therefore, this methodology will reduce the number of tests and provide a statistically significant conclusion.

4.2 Factors and Levels

Factorial ANOVA is a methodology that involves independent variables and dependent variables (DV). IVs, also known as a factor, is an element that may have several levels. The researcher manipulates the factors to observe the effects on the DVs (output) [69].

The detail of the selected factors and levels are explained as follows:

Scenes: The complexity of a scene is determined directly by the number of objects, textures, and effects [15]. For that reason, in this research, the impact of the scene complexity in the rendering time is an essential factor. So, we are going to analyze seven complex scenes:

- Barcelona Pavilion
- Bathroom
- Breakfast
- Contemporary Bathroom
- Landscape
- Crown
- Volume Caustic

Image Size: Image resolution is one of the main elements that determine the quality of an image [70]. Pixel, the smallest possible detail presented in a digital picture, is the unit to measure image resolution [17]. So, the more pixels we have in an image, the better quality it has. The advantage of the following resolutions is that the resulted images will have an aspect ratio of 16:1.

- 640x360 (nHD)
- 960x540 (qHD)
- 1280x720 (HD)

These three image resolutions will give enough quality and resolution to measure their impact on rendering time in our criteria.

Operating System: Part of the hypothesis for this research effort is the use of a special-purpose operating system to accelerate ray tracing. Also, a general-purpose OS based on Ubuntu OS will be the baseline. So, we will have the following OS:

– Special-purpose OS (based on the Yocto Project Dunfell 3.1)

– General-purpose OS (Ubuntu 18.04 LTS)

The kernel version for both OS is Linux kernel v5.4.

Scheduling Policy: This is another crucial factor for hypothesis testing. The factors are the following:

– Custom Scheduling Policy

– CFS Scheduling Policy

4.3 Measures and Combinations of the Experiment

Table 4.1 summarizes the factors and levels for the factorial ANOVA analysis. There are 84 combinations.

According to [17], a factorial ANOVA requires at least two measurements for each combination. For this research effort, six measures will be performed, granted a total of **504** cases for the ANOVA test.

The nomenclature for the table is:

Operating System: General-purpose operating system (GPOS) and special-purpose operating system (SPOS).

Scheduling Policy: CFS Scheduling Policy (CFSSP) and Custom Scheduling Policy (CSP).

4.4 Response Variable

According to [17] and [40], the performance can be measured as the CPU's time to execute a process. As mentioned at the beginning of this document, rendering time is

Table 4.1: Factors and levels of the experiment.

	Factor			
	Scene	Resolution	Operating system	Scheduling Policy
Levels	Barcelona Pavilion Bathroom Breakfast Contemporary Bathroom Crown Landscape Volume Caustic	640x360 960x540 1280x720	GPOS SPOS	CSP CFSSP

one of the most costly elements for the content creation industry. For that reason, the **rendering time** will be the response variable.

4.5 Hardware Used for the Experiments

One computer will be used for the experiments. The characteristics of the systems are shown in Table 4.2.

Table 4.2: Hardware characteristics for the experiments.

Characteristic	Description
CPU	Intel Core i7-6700HQ
Price	$378[1]
CPU clock frequency range	2.6 - 3.5 GHz
Cores/Threads	4/8
Cache L1/L2/L3	256 kB/1 MB/6 MB
Memory technology	DDR4 @ 2133 MT/s
Memory Size	12 GB (SODIMM dual-channel)
Storage (GPOS)	512 GB SSD
Storage (SPOS)	128 GB SSD
Power Consumption	45 W
Thermal solution	stock cooling

[1] This CPU was released in Q3'2015 [71]

4.6 Scenes Used for the Experiments

One of the main reasons for this research effort is to prove the potential performance improvement in rendering time for ray tracing that the industry could exploit. The complexity of a ray-traced comes from the number of objects distributed across the scenes [17]. Therefore, we will use part of the examples PBRT-v3 offers because they meet the quality and complexity that fits this research effort. The scenes are going to be modified to meet the resolution factor exposed previously. The scenes selected are:

1. Barcelona Pavilion (Figure 4.1)
2. Bathroom (Figure 4.2)
3. Breakfast (Figure 4.3)
4. Contemporary Bathroom (Figure 4.4)
5. Crown (Figure 4.5)
6. Landscape (Figure 4.6)
7. Volume Caustic (Figure 4.7)

Figure 4.1: Scene for rendering: Barcelona Pavilion [6]

Figure 4.2: Scene for rendering: Bathroom [6]

Figure 4.3: Scene for rendering: Breakfast [6]

Figure 4.4: Scene for rendering: Contemporary Bathroom [6]

Figure 4.5: Scene for rendering: Crown [6]

Figure 4.6: Scene for rendering: Landscape [6]

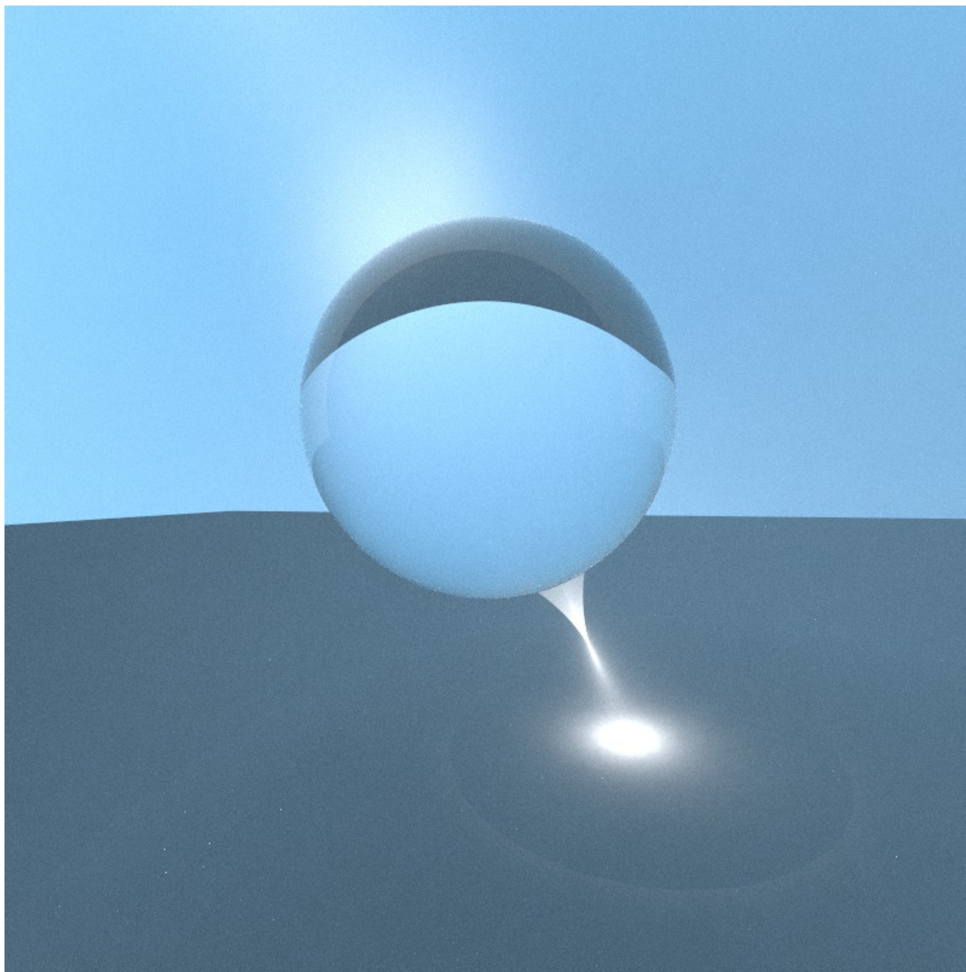


Figure 4.7: Scene for rendering: Volume Caustic [6]

4.7 Hard Cache Affinity used for the Experiments

There is an easy way to assign hard cache affinity during program execution from user-space in Linux-based OS by executing the command *taskset()* [72]. There are a few inconveniences of this approach:

1. The hard cache affinity is set when the task is already being executed.

2. The user should set the hard cache affinity manually.

3. The process would have soft cache affinity assigned for some time during its execution.

4. The reproducibility of the experiment. Using taskset() command, we are introducing a new variable to the experiment: the user. It implies that the experiment might not be completely reproduced because it depends on the user's response time.

5. Potentially, the task can suffer race conditions because artificially, the user executes another task with higher priority that preempts the ray-tracing execution [46].

6. Context switch increment due to execution of another high-priority task during the experiment. It is the opposite of what this research effort is looking to reduce [51].

The Linux-based kernel allows the addition of scheduling policies. For that reason, the outcome of this research effort will be a scheduling policy added to Linux's kernel to assign a hard cache affinity to ray tracing-related tasks. Thus, the proposed solution will be automated, sophisticated, and elegant to accelerate ray tracing through hard cache affinity. The benefits of this approach are:

1. Hard cache affinity is set before the task is added to the execution queue. The scheduling policy will fix the *cpumask* to set hard cache affinity.

2. Cache misses and context switch reduction.

3. Experiment reproducibility. As the custom scheduling policy is added to the kernel, the experiment can be easily reproduced.

4. Ray tracing execution is not stopped/preempt to assign hard cache affinity.

4.8 Rendering Library used for the Experiments

The rendering library chosen for the experiments is PBRT-V3. PBRT-V3 is a state-of-the-art library released in 2016 and a Physically Based Rendering book: From Theory to Implementation. The library is the implementation of the concepts presented in the book [55]. This book has won an Academy Award and has been the base for multiple research work, courses, and audiovisual production [15]. PBRT-v3 offers several advantages that are important for this research effort:

– It is an open-source library.

– It is supported in Linux-based environments.

– It accurately simulates materials and lights.

4.9 Special-Purpose Operating System used for Experimentation

The platform used to build the SPOS was Yocto Project. Yocto Project is an open-source community that provides templates, tools, and methods to help to create custom Linux-based systems [73]. Although Yocto Project trends to embedded systems rather than high-performance systems like desktops, it offers several advantages that fit this research effort. The advantages are:

– It is an Open Source collaboration project.

- It is supported and governed by high-tech industry leaders.
- It offers support to x86 architecture.
- Because its architecture is based on layers, making it relatively easy to integrate a new library like PBRT-v3.
- It allows kernel customization.
- There is a support platform.

The release is 3.1.1. LTS “Dunfell" is selected as the baseline to the SPOS. Linux kernel version 5.4 based on “meta-intel”. *meta-intel* is selected because it offers well-documented support for x86 architecture, and it is recommended in important community forums. For this research, a minimal image was generated.

4.10 Data Collection

A python script was developed to automate the execution and data recollection from the experiments.

4.11 Factorial Analysis of Variance (ANOVA)

Factorial ANOVA is often used to understand the combined effect of at least two different factors on a dependent variable [74]. It allows the researchers to test for group differences and their interactions [17]. The testing performed by ANOVA allows determining whether results are significant to reject the *null hypothesis* or accept the *alternate hypothesis*. In other words, ANOVA predicts if there is a difference between groups.

ANOVA assumes independence of observations and the homogeneity of variance, as per other assumptions on the general linear model [68, 74]. As a result, factorial ANOVA can have multiple groups (independent variables) and multiple levels.

The ANOVA consists of calculating the mean of each group to compare the variance of these means (inter-variability) versus the average variance within the groups (intra-variability). The null hypothesis shows that the observations come from the same population (i.e., the groups have the same mean and variance). However, as the means of the groups are further away from each other, their variance will increase. Therefore, the average variance will cease to be the same (proving that the groups are statistically different).

F_{ratio} is used to make an inference from ANOVA [68]. F_{ratio} is the ratio between the variation of the means of the groups and the average of the variation within the groups. When F_{ratio} equals '1', the null hypothesis is satisfied since the inter-variability will be the same as the intra-variability. Otherwise, the alternative hypothesis will be accepted because F_{ratio} is going to be greater than one. It means that the variance between the means of the groups will be broader compared to the average of the variance within the groups. Besides, it gives a lower probability that the population will acquire extreme values. The smaller the p-value, the more certainty there will be in accepting the alternative hypothesis [69].

4.11.1 Assumptions of the Factorial ANOVA

According to [75], these are the assumptions that must meet a factorial ANOVA:

- **Normality:** The dependent variable is normally distributed.
- **Independence:** Observations and groups are independent of each other.
- **Equality of Variance:** the variance is equal across groups.

4.11.2 Hypothesis for ANOVA

Null Hypothesis: The means of the dependent variable for each group are equal.

Alternate hypothesis: The means of the dependent variable for each group are not equal.

Chapter 5

Design

This chapter presents the design of a custom scheduling policy that assigns hard cache affinity to accelerate computationally intensive workloads such as ray tracing algorithms. It also provides the proposed changes in Linux's Kernel to implement the custom scheduling policy in a SPOS, with implementation details and limitations to the present method. In addition, the methodology to integrate this scheduling policy to the GPOS and the Yocto implementation is also shared.

5.1 Yocto Implementation

As explained above, Yocto offers developers the possibility to customize Linux distributions depending on their necessities. Following is listed the most relevant changes in Yocto:

- **PBRT implementation:** PBRT targets robust systems such as desktops, servers. A completely new recipe was developed as part of this research to implement PBRT-v3 in the SPOS. Appendix D describes the implementation of the Yocto recipe for PBRT-v3.

- **Linux Kernel:** As explained in section 4.5, the machine target has x86 architecture.

meta-intel is used as kernel baseline because it offers higher stability to the system as it controls several critical tasks such as memory access, thermal management, I/O management. Another important detail is that the kernel version used in this implementation is the release 5.4.100.

– **Unnecessary driver removed**: it pretends to improve kernel and overall system efficiency by eliminating unnecessary I/O drivers such as:

  - Bluetooth
  - WiFi
  - Graphics/display
  - Touch-pad
  - Camera

– **Additional recipes**: PBRT and the script develop to control the testing require adjacent software support to work correctly. Additional programs are:

  - Python3
  - Doxygen
  - Recipes-extended

5.2 Strategy Selected for scheduling policy implementation

As defined in the Methodology, the outcome of this research effort is a custom scheduling policy that assigns hard cache affinity. This approach was selected because it offers certain advantages over default scheduling policy such as:

1. Hard cache affinity is set before the task is added to the execution queue. The scheduling policy will fix the *cpumask* to set hard cache affinity.

2. Cache misses and context switch reduction.

3. Experiment reproducibility. As the custom scheduling policy is added to the kernel, the experiment can be easily reproduced.

4. Ray tracing execution is not stopped/preempt to assign hard cache affinity.

The main objective is to design a custom scheduling policy that assigns hard cache affinity from an implementation perspective. The scheduling policy must be flexible enough to be part of both special-purpose and general-purpose operating systems. The custom scheduling policy gives the system the power to assign hard cache affinity since the task is created by modifying the *cpumask* in the function of the number of threads created by the primary process and the number of cores available in the architecture. It will assign one thread to each core in the architecture.

During the design stage, one of the main problems was insufficient documentation about the scheduler's source code description and scheduler-to-kernel communication. So, several months were invested in understanding the kernel's source code that manages, calls, and implements the scheduler. Once the target sections of the source code were identified and taking CFS as the baseline; there were determined two main strategies to implement the custom scheduling policy.

1. **Maintaining current CFS implementation.** As it is known, the CFS is the current scheduler algorithm in Linux's Kernel. The idea here was to use the same source code provided in the kernel's distribution, but only changing the assignment of the cpumask for the cache affinity in the task descriptor would be enough to assign always the cache affinity. This strategy has the extraordinary advantage of requiring relatively minor changes (compared to strategy 2) to set hard cache affinity. Sounds promising, but it has the disadvantage that it will reduce the flexibility of the scheduler to assign soft cache affinity to a program that is not ray tracing.

2. **Writing a new source code for the scheduling policy**. This strategy looks for a completely new scheduling policy that assigns hard cache affinity. Having CFS as

a baseline, the idea behind this strategy is to write a new scheduling policy from scratch that has to be integrated with the rest of the kernel. This strategy offers the advantage to keep CFS intact and, it also allows the system to have a scheduling policy to assign hard cache affinity and another scheduling policy to assign soft cache affinity (CFS). The disadvantage of this strategy is that the implementation time takes longer than strategy 1, and it also needs more testing and debugging. By following this strategy, the integration in the GPOS for the experiment would be lesser disruptive.

The method selected is the second option because of the advantages that it offers. It also prevents the risk of modifying CFS and produces instability in the OS.

5.2.1 Writing a new source code for the scheduling policy

To write a new scheduling policy, the developer must have a deep understanding of the critical elements of the kernel that should be modified and added to ensure correct and stable implementation.

Custom scheduling policy (CSP) will have the maximum priority (even higher than real-time tasks). The intention of assigning higher priority is to run PBRT tasks with more stability. Figure 5.1 shows a high-level flowchart for the CSP. When the user sends to execute the ray tracing task, the Linux kernel will assign and create the data structures required to execute and manage the task. The kernel will save all the parameters of the task in a data struct named *task_struct*. In the *task_struct*, we added a parameter to identify if the task is a PBRT job.

As CSA has the maximum priority, there is a potential risk of mixing non-PBRT jobs with PBRT jobs. A security check was implemented to prevent kernel instability by accidentally assigning CSA to a non-target task. If the task has *SCHED_CUSTOM_POLICY* as *sched_class*, when the task is ready to enter in the *sched_class run queue*, the CSP checks whether the task is a PBRT task. If yes, the

parameter *is_pbrt* (a bool type variable in the custom_task data structure) changes to '1' (*true*). Then, the CSP assigns hard cache affinity to each thread in the task. Finally, the task is added in the run queue (double-linked list and RBTREE).

If the task is not a PBRT task, the task will not be added to the CSP run queue. Then, CSP will modify the scheduling policy assigned to the task depending on its priority. If the priority is zero, the new scheduling policy will be *SCHED_NORMAL*. Otherwise, the new scheduling policy is *SCHED_FIFO* because the task is an RT job. Finally, the scheduler will continue the normal execution.

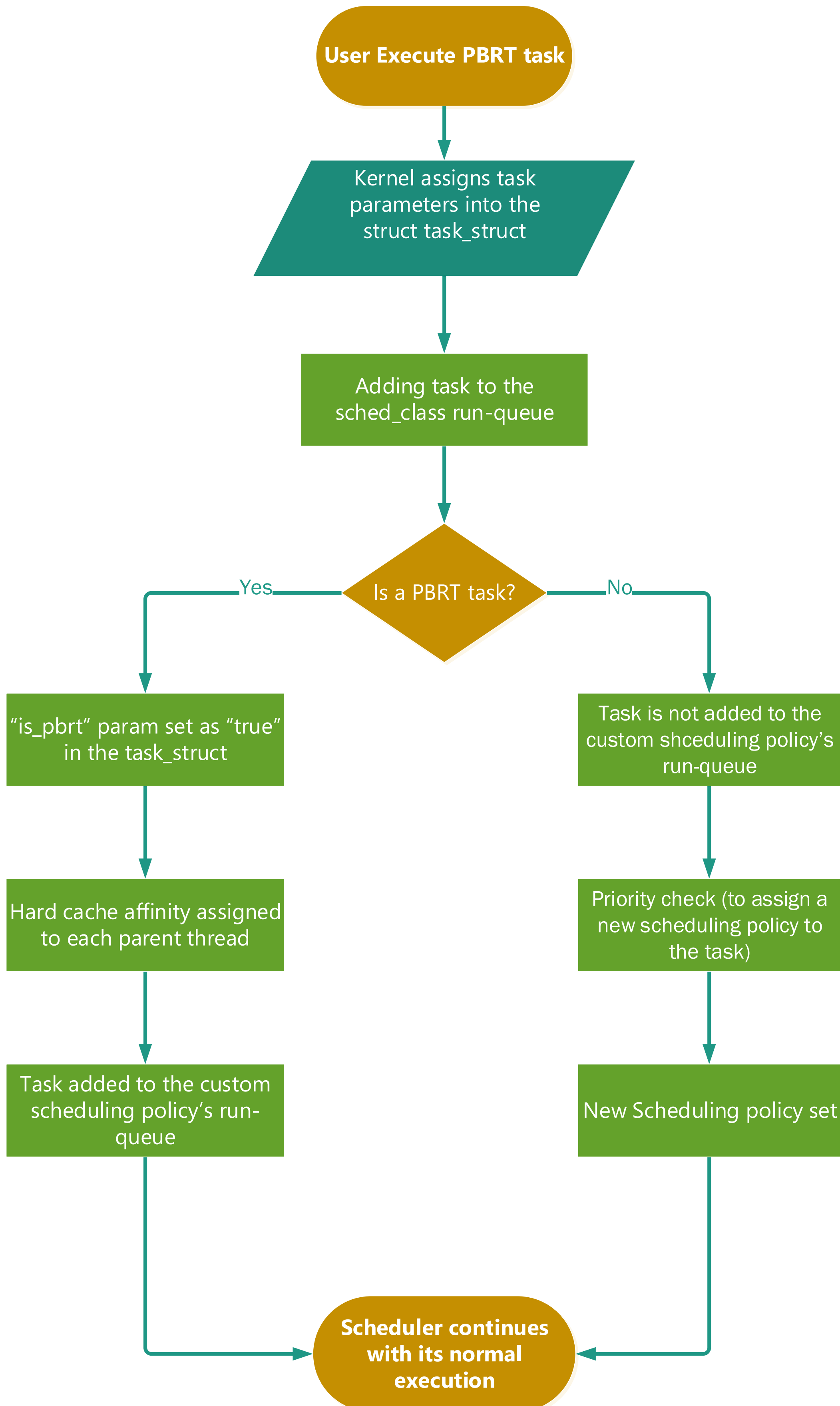


Figure 5.1: Flowchart for the custom scheduling policy

5.2.2 Source Code

Many changes, design and source-code addition to tje Linux's kernel was required to implement the custom scheduling policy. Appendix A presents the most relevant files modified and added as part of this research to design the new scheduling policy named as **custom_scheduler**.

5.3 Adding the custom scheduling policy in the GPOS

Appendix C shows the methodology followed to recompile Ubuntu's kernel to add the custom scheduling policy. It was required to ensure a fair comparison in the experiment when a special-purpose operating system with a custom scheduling policy is compared against the general-purpose operating system with a custom scheduling policy.

5.4 Limitations and Requirements

As previously mentioned, the proposed design aims to support PBRT tasks, which means that any other workload may need additional modifications. The changes listed above in Linux's Kernel source code are valid for Kernel release 5.4.100 and Yocto Project 3.1.1 Dunfell. Another critical assumption is that the custom scheduling policy must have the maximum priority to ensure PBRT tasks uses the correct scheduling policy. Any additional experiment out of the scope of this research may require more changes in the Kernel's source code.

Chapter 6

Results

This chapter presents the results obtained from the evaluation performed to compare the performance gain due to custom scheduling policy in a GPOS against CFS scheduling policy, SPOS with custom scheduling policy against SPOS with CFS scheduling policy, and a comparison of a SPOS against GPOS. Besides, at the end of this chapter, the factorial ANOVA experiment is presented. All the statistical results in this chapter were obtained using the JMP tool.

6.1 Cache optimization observed due to the custom scheduling policy

As defined in the hypothesis of this research effort, the cache misses due to the context switching will increase the execution time of computationally intense workloads such as ray tracing. Using Linux based perf profiler tool, the statistics of cache-misses, context-switches, and processor migration were obtained [76]. The profiling were applied in the GPOS with custom scheduling policy and CFS scheduling policy for a relatively simpler scene at resolution of 1280x720. Fifteen iterations were performed.

As defined in the hypothesis of this research effort, the cache misses due to the context

switching will increase the execution time of computationally intense workloads such as ray tracing. Using Linux based perf profiler tool, the statistics of cache-misses, context-switches, and processor migration were obtained [76]. The profiling was applied in the GPOS with a custom scheduling policy and CFS scheduling policy for a 'simpler' scene at a resolution of 1280x720.

Table 6.1 shows the average data obtained after profiling. Cache-misses decreased when custom scheduling policy from 13.26 billion to 12.13 billion of cache misses which is a 16% of improvement. Context-switch is another important statistic that decreased from 32 619 to 21 957 when custom scheduling policy is applied to ray tracing execution. Finally, migration means the number of times the parent thread moves its execution to another CPU. As observed in the table, the parent's migration for custom scheduling policy is zero, which confirms that the custom scheduling policy applies hard cache affinity as expected. Furthermore, the reduction of cache-misses and context-switch can also be explained by the reduction in migration because the thread does not change the CPU to improve the cache performance. As expected in this experiment, the rendering time was also reduced by approximately 9%.

Table 6.1: Profiling results

Scheduling Policy	**Profiling Statistics**			
	Cache-misses (billions)	Context-switches	Rendering Time (s)	Migrations
CFS Scheduling Policy	13.26	32619	167.73	353
Custom Scheduling Policy	11.13	21957	152.24	0
Improvement	**16.06%**	**32.69%**	**9.24%**	**NA**

6.2 Performance comparison of custom scheduling policy against other ANOVA factor combinations

As defined in the specific objectives, a performance comparison is performed to evaluate the impact of assigning the custom scheduling policy in a SPOS against the CFS scheduling policy in a SPOS, the custom scheduling policy in a GPOS, and the CFS

scheduling policy in a GPOS. The average means were obtained with six repetitions for each case. These results are provided as preliminary results before the ANOVA analysis that will provide the evidence to decide whether or not the results are statistically valid. Appendix B.1 shows the data gathered after the experiments.

Mean plots can be an initial visual method to estimate the difference between factors [77, 78]. Figure 6.1 shows the mean plot for rendering time in function of the scheduling policy with a combination of all other factors (OS, resolution, and scenes). The trend indicates that we have a rendering time reduction by applying the custom scheduling policy.

Figure 6.1: Mean plot for rendering time in function of the scheduling policy

Figure 6.2 displays the mean plot for rendering time in function of the OS with a combination of all other factors (scheduling policy, resolution, and scenes). The trend indicates a reduction of rendering time due to the SPOS.

Figure 6.3 present the mean plot for rendering time in function of all the factors

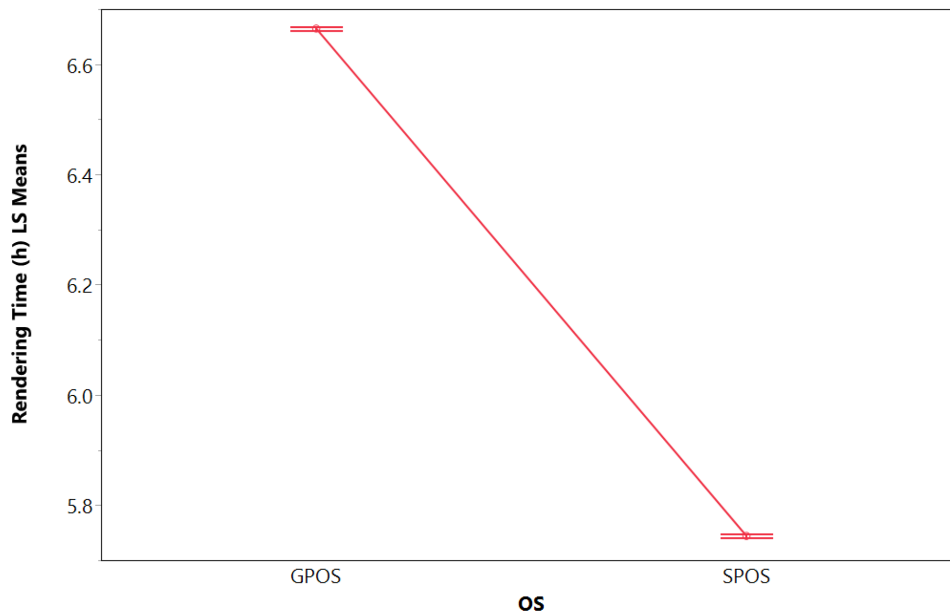


Figure 6.2: Mean plot for rendering time in function of the OS

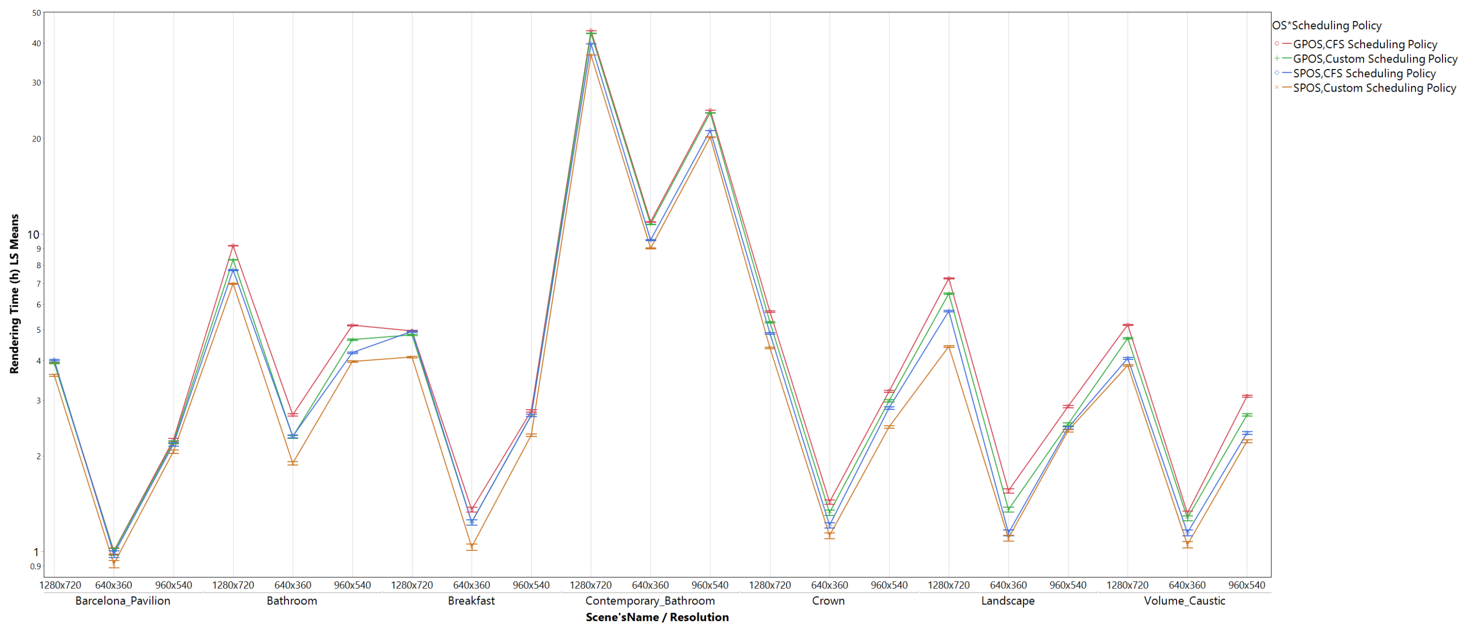


Figure 6.3: Mean plot for rendering time in function of the ANOVA's factors

(OS, scenes, resolution, and scheduling policy). As can be seen, the combination of SPOS-custom scheduling policy presents a lower rendering time compared with the other combinations. Something important to point is that the y-axis scale is logarithmic due to the big mean difference introduced mainly by the 'Contemporary Bathroom' scene.

Figure 6.4 presents a subset of data from Figure 6.3. This subset intends to show the reduction of the execution time in detail due to the combination of SPOS and custom

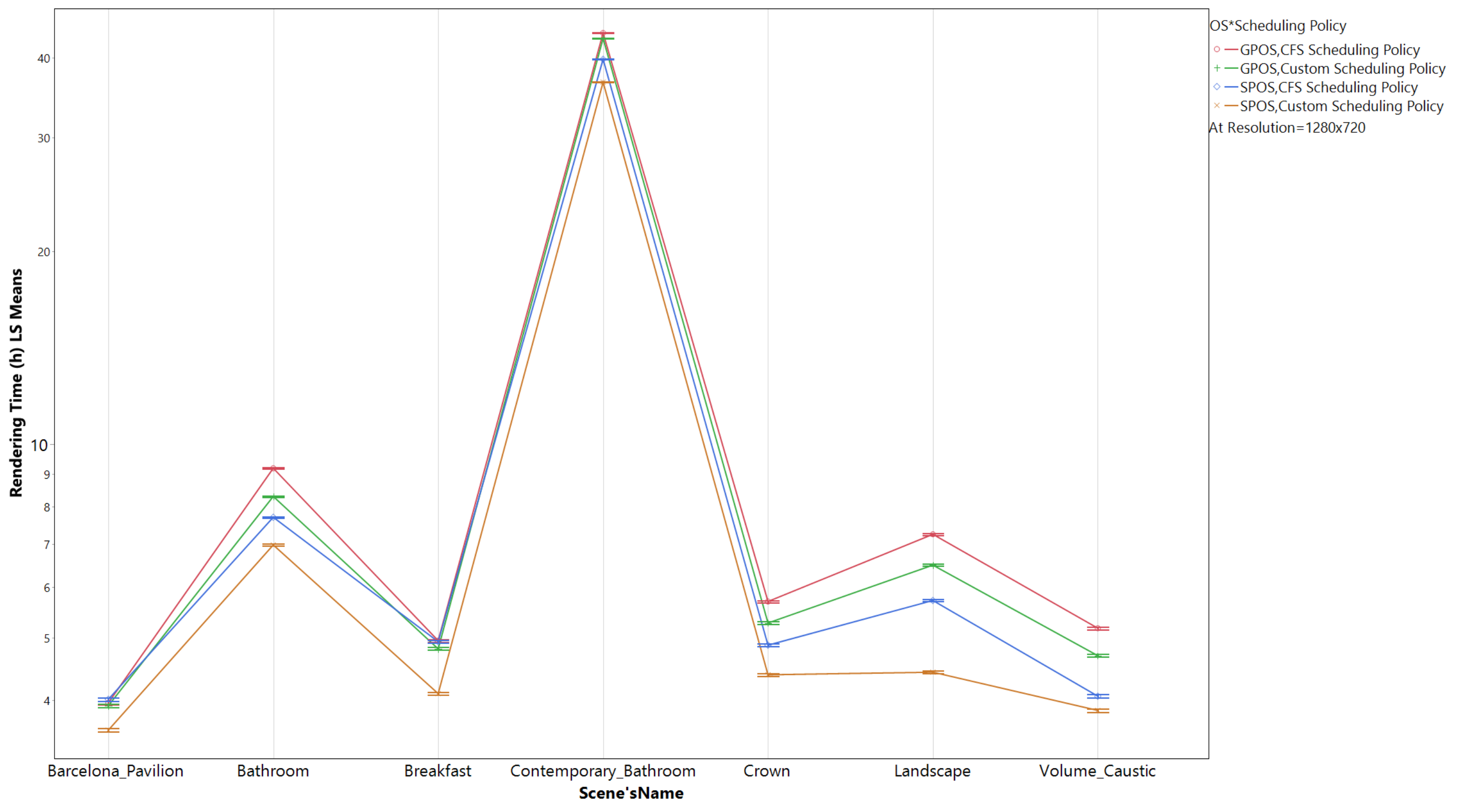


Figure 6.4: Mean plot for rendering time in function of the OS, scheduling policy, and scenes with maximum resolution

scheduling policy. This figure has rendering time in function of OS, scheduling policy, and scenes with a resolution of 1280x720. Like Figure 6.3, the y-axis CFSSPle is logarithmic due to the big mean difference introduced by the 'Contemporary Bathroom' scene.

Figure 6.4 presents a subset of data of Figure 6.3. This subset intends to show the execution time reduction in detail due to the combination of SPOS and custom scheduling policy. The conditions in this figure are: rendering time in function of the OS, scheduling policy, and scenes with a resolution of 1280x720. Like Figure 6.3, the y-axis scale is logarithmic due to the big mean difference introduced by the 'Contemporary Bathroom' scene.

6.3 ANOVA Assumptions Tests

As mentioned by [77] and [79], data quality must be checked before running any ANOVA analysis. Data quality is met by fulfilling the assumptions for ANOVA explained in the Methodology Chapter.

- **Independence**

  Observations and groups are independent each other. The independence can be tested by running a probability test between each factor to probe that. However, as [78] explains, the independence can be assessed by analyzing the experiment's design, which is met in the experiment conducted in this research.

- **Normality and homogeneity of variance**

  The dependent variable is normally distributed. The easiest way to check the normality of the data is to do a visual analysis by plotting the residual against the normal quantile [80, 81]. Figure 6.5 shows that the data is not following a normal distribution because, in a normal distribution, the points should follow the best fit line (red line). A formal test to confirm the normality of the data is presented in Figure 6.6. Shapiro-Wilk W Test indicates how normally distributed is a set of data. A *p-value* $<$ *0.05* indicates a violation of the assumption of normality [82]. The results show a p-value below 0.05, which confirms that the data is not normally distributed.

The next step is to check the homogeneity of variance. The residual plot works to do a visual review of the homogeneity. According to [16], the residual in Figure 6.7 should have the same spread across all the fitted values to confirm the homogeneity of variance, which is rejected as the residuals are not uniformly distributed. Besides the visual review, Levene's test is the formal method chosen to test the homogeneity of variance. Figure 6.8 shows two critical results:

1. Standard Deviation plot in the function of each factor (OS, resolution, scheduling

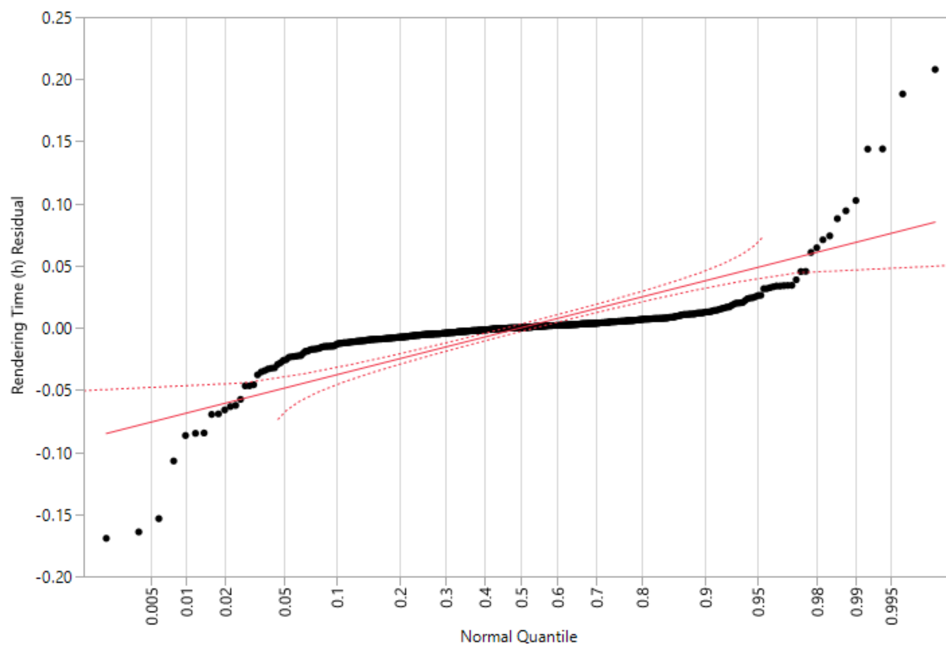


Figure 6.5: Residual in function of the Normal Quantile

Figure 6.6: Residual normality test for rendering time data.

policy, and scene). There are a few outliers introduced by the scene *Contemporary Bathroom* that affects the homogeneity of variance assumption.

2. Levene's test result. As shown in the figure, there are results for other tests like

O'Brien[.5], Brown-Forsythe, and Bartlett. For the present results, we rely on Levene's test due to its robustness [80, 81]. Levene's test indicates that the data has homogeneity of variance for values *p>0.05* (i.e., rejecting the Null Hypothesis). It means that the data for rendering time is the heterogeneity of variance.

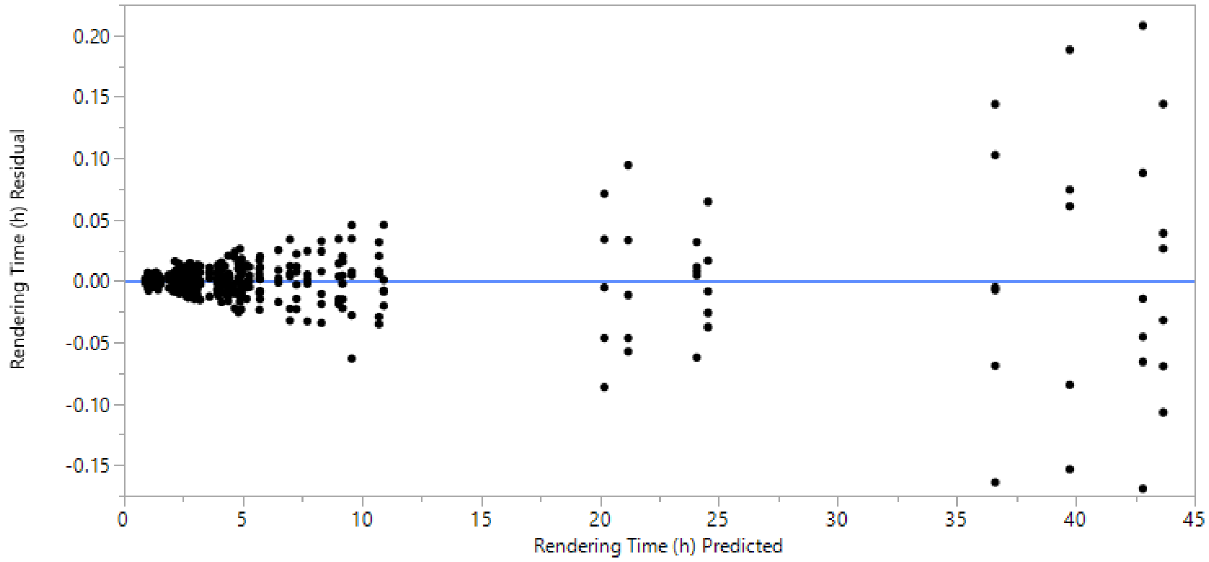
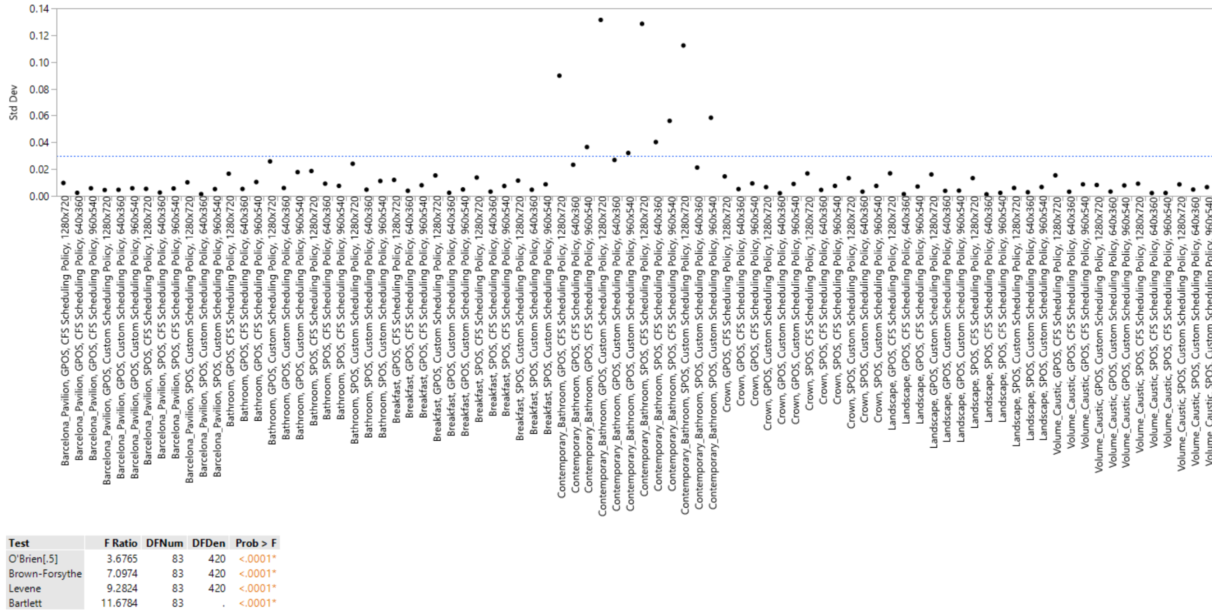


Figure 6.7: Residual by predicted plot



Test	F Ratio	DFNum	DFDen	Prob > F
O'Brien[.5]	3.6765	83	420	<.0001*
Brown-Forsythe	7.0974	83	420	<.0001*
Levene	9.2824	83	420	<.0001*
Bartlett	11.6784	83	.	<.0001*

Figure 6.8: Homogeneity of variance

For scenarios where both assumptions are violated, [16], [82], and [83] recommend generating a transformation to the output variable. The transformation is a simple method to present the data, so it should not affect the conclusion.

Box-Cox transformations are applied to the data. Box-Cox transformation is recommended to normalize the behavior of the data. Figure 6.9 shows the results of the Box-Cox transformation, which is a constant (λ) to scale the data to obtain the best normal fit. With the new data, the ANOVA assumptions are rechecked:

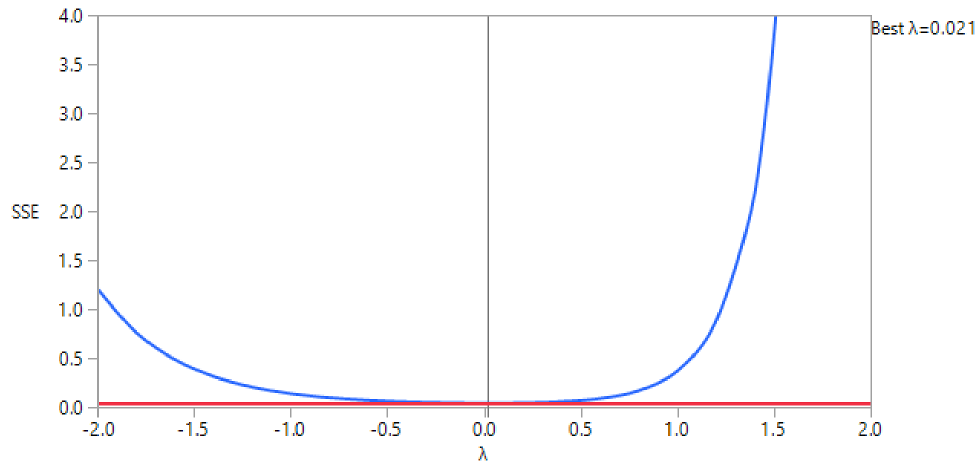


Figure 6.9: Box-Cox transformation result.

- **Normality**

  Doing a visual review of Figure 6.10 seems that the residual follows the red line (fit line), closely than Figure 6.5. Hence, the transformation improved the normality of the data. It is confirmed by analyzing the result of the *Shapiro-Wilk test* in Figure 6.11, where a result of $p - value = 0.2573$ (> 0.05) confirms that the assumption of normality is not violated.

- **Homogeneity of variance**

  Figure 6.12 shows that the residual behavior keeps a uniform spread around the zero value for the residual axis after the transformation. Figure 6.13 confirms the improvement in the homogeneity of variance, firstly observing a more uniform behavior of the standard deviation, which is confirmed by Levene's test that had a result of the $p - value = 0.3795$ (> 0.05).

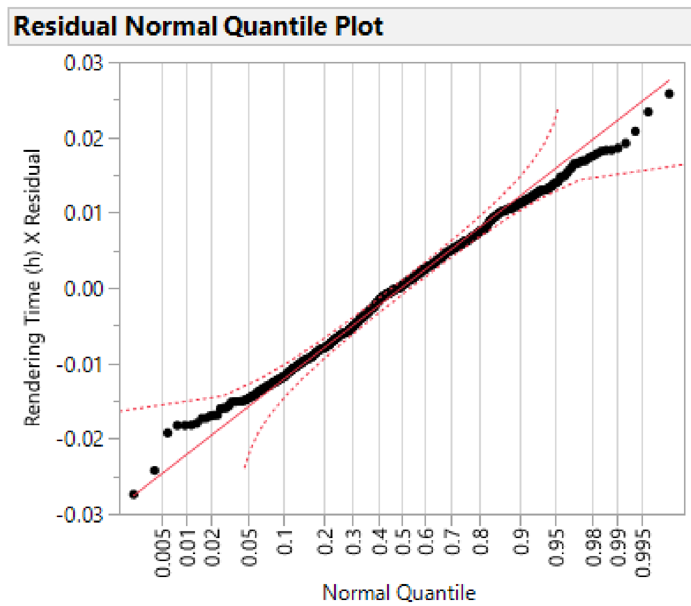


Figure 6.10: Residual in function of the Normal Quantile with rendering time transformed

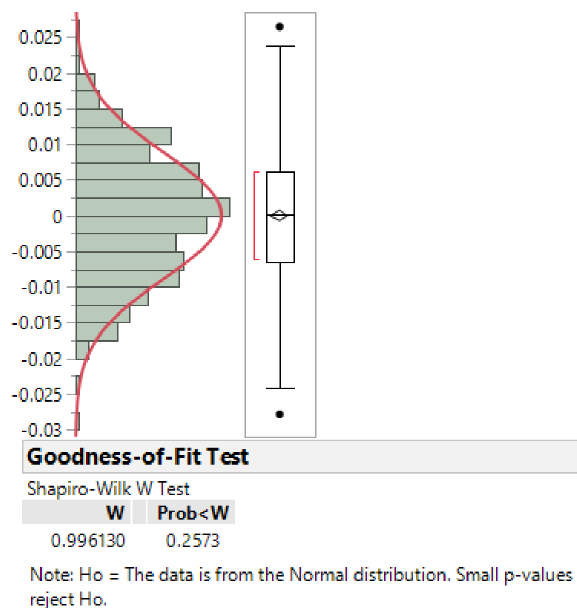


Figure 6.11: Residual normality test for transformed rendering time data.

6.4 ANOVA Results

ANOVA was performed with the Box-Cox transformed data to meet with ANOVA assumptions requirements. An important disclaimer is that the following data was generated and processed with a Box-Cox ($\lambda = 0.021$) transformation for the response variable. Besides, all the presented mean in the charts are the back-transformed data results ob-

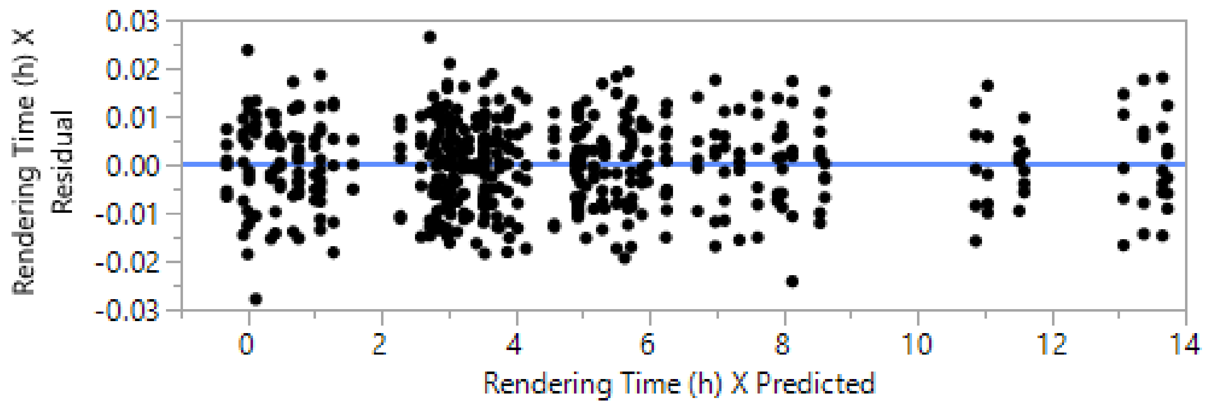
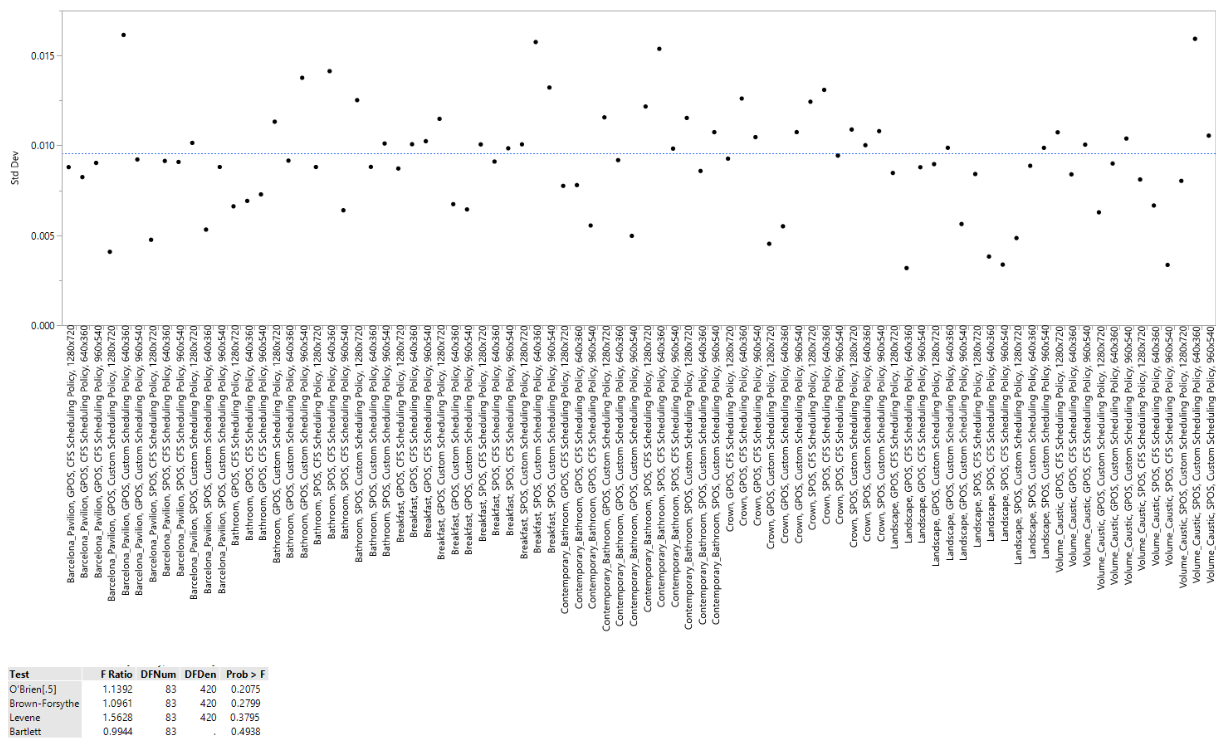


Figure 6.12: Residual by predicted plot with data transformed



Test	F Ratio	DFNum	DFDen	Prob > F
O'Brien[.5]	1.1392	83	420	0.2075
Brown-Forsythe	1.0961	83	420	0.2799
Levene	1.5628	83	420	0.3795
Bartlett	0.9944	83	.	0.4938

Figure 6.13: Homogeneity of variance with data transformed

tained from the analysis. Figure 6.14 present the results of the ANOVA effect test. The first column (Source) indicates the factors and their combinations. The penultimate column indicates the F-ratio, and the last column indicates the p-value for each factor and combination. The F-ratio column helps to understand how good (or bad) our model is. As can be seen, all the F-ratio column values are pretty big numbers, which is a good indication that we have a good model.

As the figure shows, all the factor and their combinations have a significance level

($p - value << 0.05$), so undoubtedly, there is evidence that group means differ from each other.

Effect Tests					
Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
Scene'sName	6	6	3764.9436	6850674	<.0001*
OS	1	1	37.4102	408428.8	<.0001*
Scheduling Policy	1	1	11.8266	129118.0	<.0001*
Resolution	2	2	2032.3157	11093977	<.0001*
Scene'sName*OS	6	6	5.0744	9233.376	<.0001*
Scene'sName*Scheduling Policy	6	6	1.4614	2659.169	<.0001*
Scene'sName*Resolution	12	12	13.0698	11890.91	<.0001*
OS*Scheduling Policy	1	1	0.3812	4162.105	<.0001*
OS*Resolution	2	2	0.1724	940.8255	<.0001*
Scheduling Policy*Resolution	2	2	0.1416	773.0851	<.0001*
Scene'sName*OS*Scheduling Policy	6	6	0.9123	1660.095	<.0001*
Scene'sName*OS*Resolution	12	12	2.0488	1864.016	<.0001*
Scene'sName*Scheduling Policy*Resolution	12	12	0.9949	905.1150	<.0001*
OS*Scheduling Policy*Resolution	2	2	0.2585	1410.893	<.0001*
Scene'sName*OS*Scheduling Policy*Resolution	12	12	0.9166	833.9120	<.0001*

Figure 6.14: ANOVA effect test result

Tukey's HSD (honestly significant difference) test is selected as *post hoc* analysis. Tukey's test calculates a new critical value that can be used to evaluate whether or not differences between any two pairs of means are significant while controlling the probability of making one or more Type I errors [80, 84, 85]. The most critical experiment factor to test the hypothesis of this research is the scheduling policy. Specifically, if the group means of the combination SPOS-custom scheduling policy against SPOS-CFS scheduling policy, GPOS-custom scheduling policy, and GPOS-CFS scheduling policy is statistically different. Tukey's HSD test was used to test the mean differences in the rendering time between groups through pairwise comparisons. As standard, the significance value of this test is 0.05 (α), so any p-value less than this value indicates a statistical difference between the rendering time of the groups.

Table 6.3 to Table 6.6 presents the connecting letters report where shared letters indicate no difference between groups, while different letters indicate a statistical difference.

Besides, the p-value is also presented where green represents that the group's means are statistically different and red represents that they are not statistically different.

Table 6.6 only presents the data for the resolution of 1280x720 because presenting Tukey's HSD result for all the combinations imply a big table. The nomenclature for the effects is presented in Table 6.2.

Table 6.2: Nomenclature for Tukey's HSD tables

Factor	Nomenclature	Meaning
Operating System	SPOS	Special-purpose Operating System
	GPOS	General-purpose Operating System
Scheduling Policy	CSP	Custom Scheduling Policy
	CFSSP	CFS Scheduling Policy
Scene's Name	S1	Barcelona Pavilion
	S1	Bathroom
	S3	Breakfast
	S4	Contemporary Bathroom
	S5	Crown
	S6	LandCFSSPpe
	S7	Volume Caustic
Resolution	R1	1280x720
	R2	960x540
	R3	640x360

Table 6.3: Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP

(a) P-value table

	SPOS-CSP
SPOS-CFSSP	0
GPOS-CFSSP	0
GPOS-CSP	0

(b) Connecting letters report

Level					Least Sq Mean
GPOS,CFSSP	A				4.0539047
GPOS,CSP		B			3.7793828
SPOS,CFSSP			C		3.5363241
SPOS,CSP				D	3.1963716

Example: SPOS:CSP:R1:S1 is a case that runs in the special-purpose operating system with the custom scheduling policy, resolution of 1280x720, and the scene is Barcelona Pavilion.

Tukey's HSD test allows us to know if the average rendering time of the combination SPOS/CSP against the other factors/levels is statistically different. However, it does not

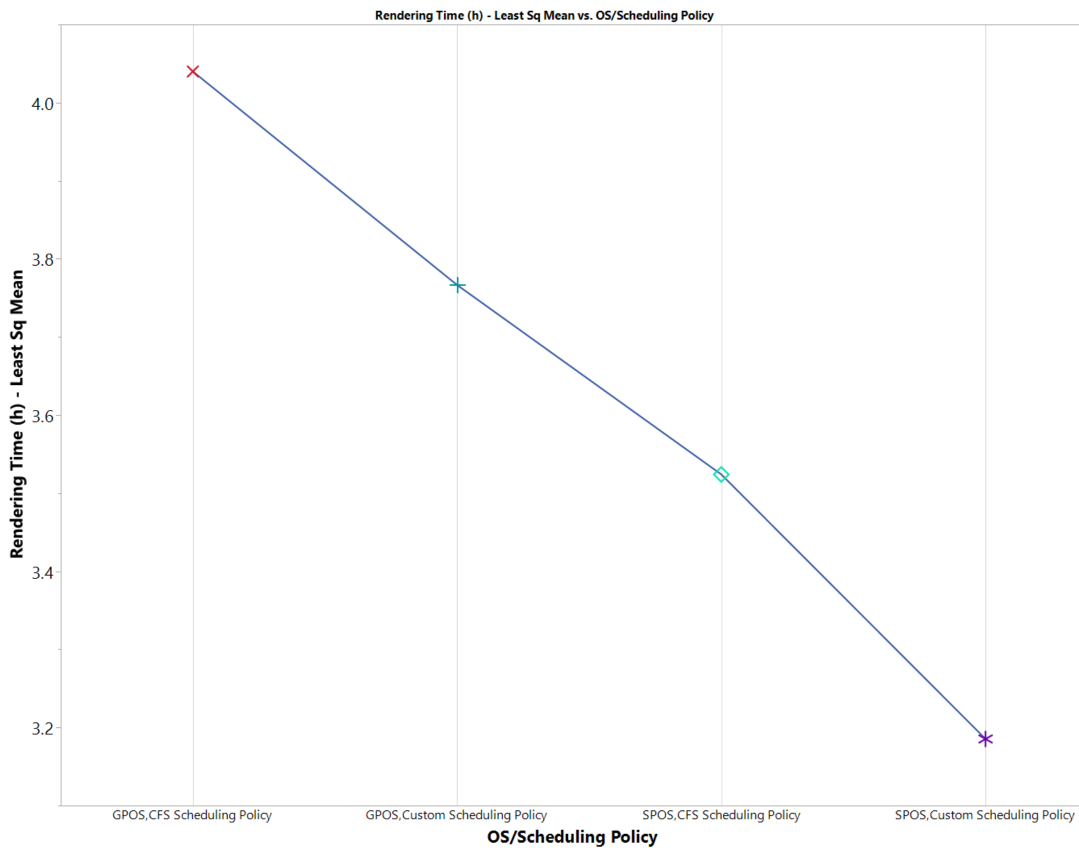


Figure 6.15: Mean plot for Rendering Time (h) in function of the OS/scheduling policy

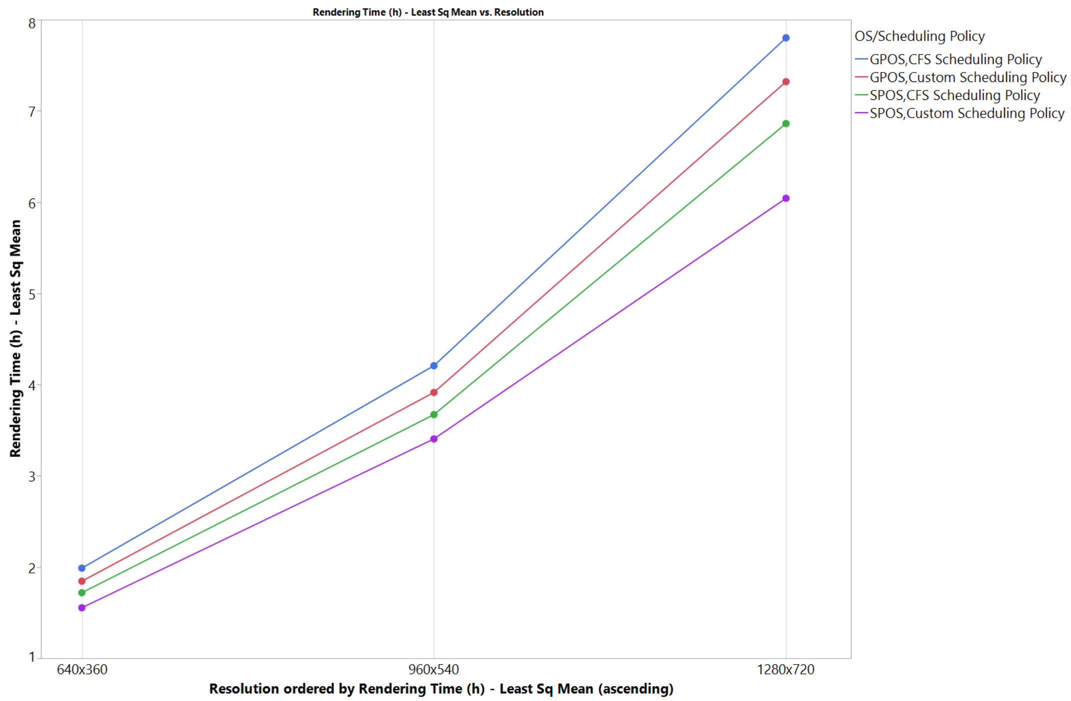


Figure 6.16: Mean plot for Rendering Time (h) in function of the OS/scheduling policy with sweep of image resolution

Table 6.4: Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with three resolution scenes

(a) P-value table

	SPOS-CSP:1280x720	**SPOS-CSP:960x540**	**SPOS-CSP:640x360**
SPOS-CFSSP:1280x720	0		
SPOS-CFSSP:960x540		0	
SPOS-CFSSP:640x360			0
GPOS-CFSSP:1280x720	0		
GPOS-CFSSP:960x540		0	
GPOS-CFSSP:640x360			0
GPOS-CSP:1280x720	0		
GPOS-CSP:960x540		0	
GPOS-CSP:640x360			0

(b) Connecting letters report

Level													Least Sq Mean
GPOS,CFSSP,1280x720	A												7.1904764801
GPOS,CSP,1280x720		B											6.7474553697
SPOS,CFSSP,1280x720			C										6.3254437338
SPOS,CSP,1280x720				D									5.5694225786
GPOS,CFSSP,960x540					E								4.1979438279
GPOS,CSP,960x540						F							3.9055952395
SPOS,CFSSP,960x540							G						3.6635653426
SPOS,CSP,960x540								H					3.3979007546
GPOS,CFSSP,640x360									I				2.1970350321
GPOS,CSP,640x360										J			2.0389634328
SPOS,CFSSP,640x360											K		1.8993855466
SPOS,CSP,640x360												L	1.7178386008

let us know if the SPOS/CSP presents the minimum rendering time. The Least Squares Means Plot presented from Figure 6.15 to Figure 6.18 in conjunction with Tables 6.3 to 6.6 lets us infer this information.

All these figures suggest that no matter the factor and level analyzed, the combination of SPOS with CSP gives the maximum performance. This statement ceases to be anecdotal and becomes statistically valid when analyzed with Tukey's HSD results. As observed, for example, Figure 6.18 demonstrates that it does not matter the combination of scene and resolution, the SPOS:CSP produces lower rendering time. It can be confirmed by observing Table 6.6, where the p-value for each combination is zero, which allows us to conclude with statistical validity that SPOS:CSP performs better when compared against the other combinations of OS/scheduling policy.

Table 6.5: Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with seven scenes

(a) P-value table

	SPOS:CSP:S1	**SPOS:CSP:S2**	**SPOS:CSP:S3**	**SPOS:CSP:S3**	**SPOS:CSP:S5**	**SPOS:CSP:S6**	**SPOS:CSP:S7**
SPOS:CFSSP:S1	0						
SPOS:CFSSP:S2		0					
SPOS:CFSSP:S3			0				
SPOS:CFSSP:S3				0			
SPOS:CFSSP:S5					0		
SPOS:CFSSP:S6						0	
SPOS:CFSSP:S7							0
GPOS:CFSSP:S1	0						
GPOS:CFSSP:S2		0					
GPOS:CFSSP:S3			0				
GPOS:CFSSP:S3				0			
GPOS:CFSSP:S5					0		
GPOS:CFSSP:S6						0	
GPOS:CFSSP:S7							0
GPOS:CSP:S1	0						
GPOS:CSP:S2		0					
GPOS:CSP:S3			0				
GPOS:CSP:S3				0			
GPOS:CSP:S5					0		
GPOS:CSP:S6						0	
GPOS:CSP:S7							0

(b) Connecting letters report

Level																										**Least Sq Mean**
Contemporary_Bathroom,GPOS,CFSSP	A																									22.786544119
Contemporary_Bathroom,GPOS,CSP		B																								22.350991817
Contemporary_Bathroom,SPOS,CFSSP			C																							20.124381015
Contemporary_Bathroom,SPOS,CSP				D																						18.890187967
Bathroom,GPOS,CFSSP					E																					5.0493913034
Bathroom,GPOS,CSP						F																				4.4755340099
Bathroom,SPOS,CFSSP							G																			4.2320622154
Bathroom,SPOS,CSP								H																		3.7560660261
LandCFSSPpe,GPOS,CFSSP									I																	3.1985109378
Crown,GPOS,CFSSP										J																2.9776158901
LandCFSSPpe,GPOS,CSP											K															2.8219836471
Volume_Caustic,GPOS,CFSSP												L														2.7740206985
Crown,GPOS,CSP													M													2.7634468237
Breakfast,GPOS,CFSSP														N												2.6593559715
Crown,SPOS,CFSSP															O											2.5664923031
Breakfast,SPOS,CFSSP																P										2.5510817249
LandCFSSPpe,SPOS,CFSSP																	Q									2.5351953338
Volume_Caustic,GPOS,CSP																	Q									2.533902939
Breakfast,GPOS,CSP																	Q									2.5275988422
Crown,SPOS,CSP																		R								2.307896479
LandCFSSPpe,SPOS,CSP																			S							2.279323114
Volume_Caustic,SPOS,CFSSP																				T						2.2296862142
Breakfast,SPOS,CSP																					U					2.1523924142
Volume_Caustic,SPOS,CSP																						V				2.0883511058
Barcelona_Pavilion,GPOS,CFSSP																						V				2.0838513815
Barcelona_Pavilion,GPOS,CSP																							W			2.0621698492
Barcelona_Pavilion,SPOS,CFSSP																								X		2.0549483074
Barcelona_Pavilion,SPOS,CSP																									Y	1.9007179334

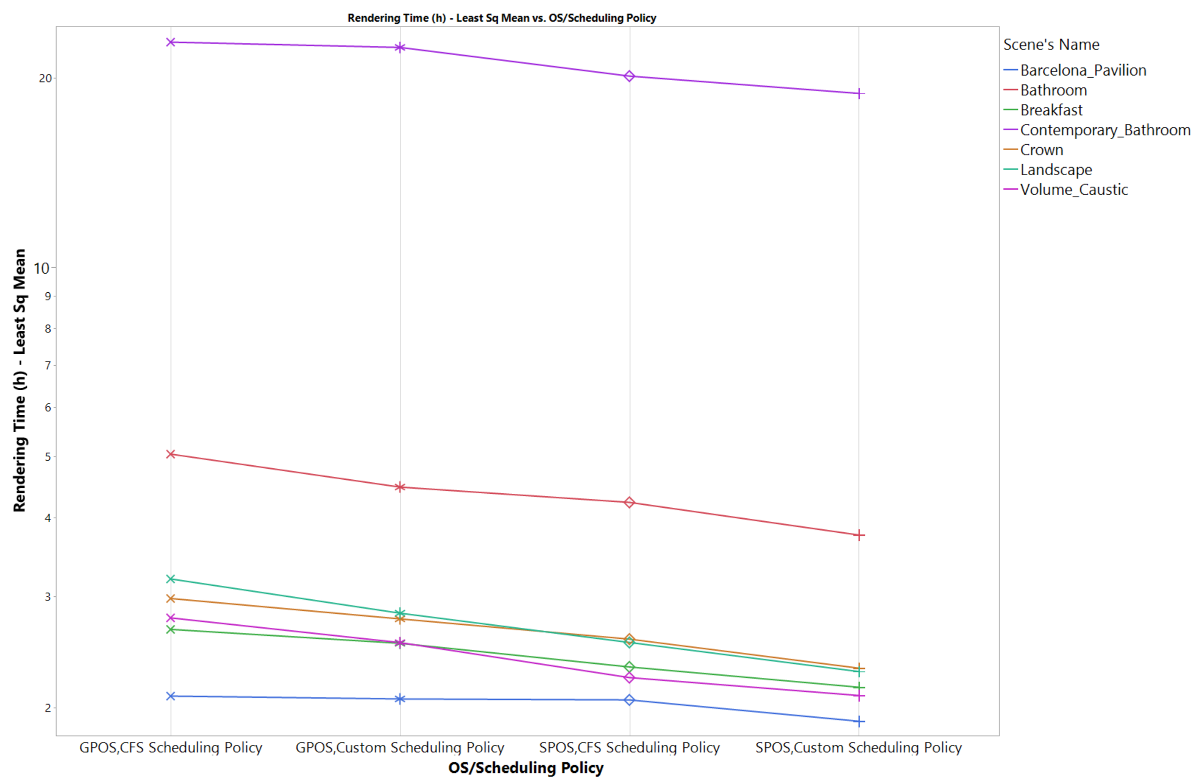


Figure 6.17: Mean plot for Rendering Time (h) in function of the OS/scheduling policy with different scenes

Table 6.6: Tukey's HSD test of the SPOS-CSP against SPOS-CFSSP, GPOS-CSP, and GPOS-CFSSP with seven scenes at resolutions 1280x720

(a) P-value table

	SPOS:CSP:R1:S1	**SPOS:CSP:R1:S2**	**SPOS:CSP:R1:S3**	**SPOS:CSP:R1:S3**	**SPOS:CSP:R1:S5**	**SPOS:CSP:R1:S6**	**SPOS:CSP:R1:S7**
SPOS:CFSSP:R1:S1	0						
SPOS:CFSSP:R1:S2		0					
SPOS:CFSSP:R1:S3			0				
SPOS:CFSSP:R1:S3				0			
SPOS:CFSSP:R1:S5					0		
SPOS:CFSSP:R1:S6						0	
SPOS:CFSSP:R1:S7							0
GPOS:CFSSP:R1:S1	0						
GPOS:CFSSP:R1:S2		0					
GPOS:CFSSP:R1:S3			0				
GPOS:CFSSP:R1:S3				0			
GPOS:CFSSP:R1:S5					0		
GPOS:CFSSP:R1:S6						0	
GPOS:CFSSP:R1:S7							0
GPOS:CSP:R1:S1	0						
GPOS:CSP:R1:S2		0					
GPOS:CSP:R1:S3			0				
GPOS:CSP:R1:S3				0			
GPOS:CSP:R1:S5					0		
GPOS:CSP:R1:S6						0	
GPOS:CSP:R1:S7							0

(b) Connecting letters report

Level																											**Least Sq Mean**
Contemporary_Bathroom,GPOS,CFSSP,1280x720	A																										43.675943333
Contemporary_Bathroom,GPOS,CSP,1280x720		B																									42.829187638
Contemporary_Bathroom,SPOS,CFSSP,1280x720			C																								39.751080784
Contemporary_Bathroom,SPOS,CSP,1280x720				D																							36.619350291
Bathroom,GPOS,CFSSP,1280x720					L																						9.1864229004
Bathroom,GPOS,CSP,1280x720						N																					8.3037636212
Bathroom,SPOS,CFSSP,1280x720							O																				7.706879771
LandCFSSPpe,GPOS,CFSSP,1280x720								P																			7.2505394906
Bathroom,SPOS,CSP,1280x720									Q																		6.9788087488
LandCFSSPpe,GPOS,CSP,1280x720										R																	6.4927615944
LandCFSSPpe,SPOS,CFSSP,1280x720											S																5.7206818178
Crown,GPOS,CFSSP,1280x720											S																5.6995217753
Crown,GPOS,CSP,1280x720												T															5.2770799458
Volume_Caustic,GPOS,CFSSP,1280x720													U														5.1798192943
Breakfast,GPOS,CFSSP,1280x720														V													4.9538307755
Breakfast,SPOS,CFSSP,1280x720														V													4.9354473003
Crown,SPOS,CFSSP,1280x720															W												4.8742355919
Breakfast,GPOS,CSP,1280x720																X											4.810350462
Volume_Caustic,GPOS,CSP,1280x720																	Y										4.6945312166
LandCFSSPpe,SPOS,CSP,1280x720																		A1									4.4249967219
Crown,SPOS,CSP,1280x720																			B1								4.3840576811
Breakfast,SPOS,CSP,1280x720																				D1							4.0980887282
Volume_Caustic,SPOS,CFSSP,1280x720																					E1						4.0603156288
Barcelona_Pavilion,SPOS,CFSSP,1280x720																						F1					4.0149508329
Barcelona_Pavilion,GPOS,CFSSP,1280x720																							G1				3.9655458412
Barcelona_Pavilion,GPOS,CSP,1280x720																								H1			3.9251368139
Volume_Caustic,SPOS,CSP,1280x720																									I1		3.8580129474
Barcelona_Pavilion,SPOS,CSP,1280x720																										J1	3.5951271276

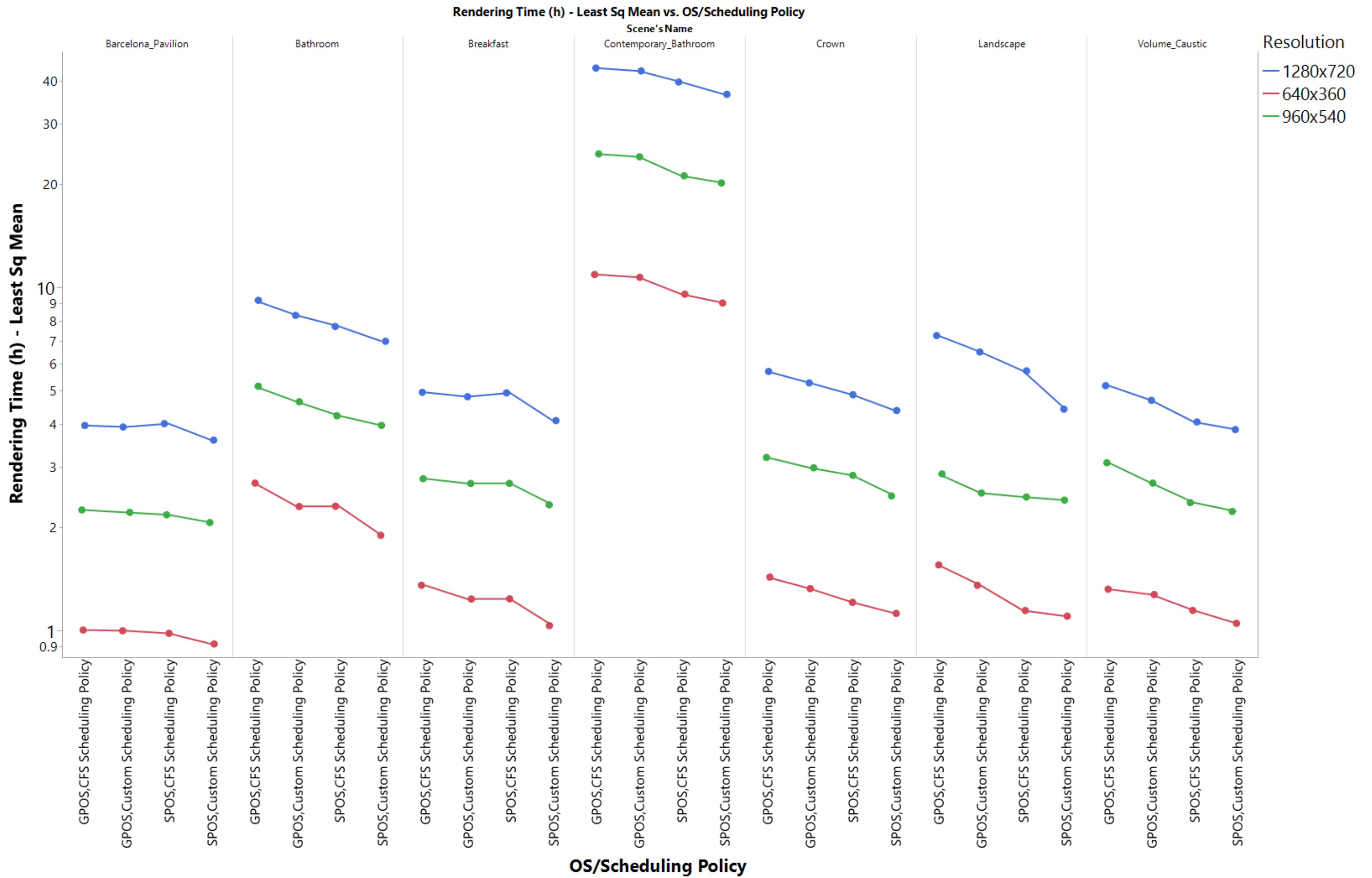


Figure 6.18: Mean plot for Rendering Time (h) in function of the OS/scheduling policy with a combination of different scenes and image resolution

6.5 Obtained Metrics

Table 6.7 shows the performance metrics obtained for SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP for all the combinations of all scenes and resolutions. For the three metrics, the combination of OS/scheduling policy with the lower value is the one that has better performance.

Table 6.7: Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP (lower is better)

Metric	SPOS:CSP	SPOS:CFSSP	GPOS:CSP	GPOS:CFSSP
Average Rendering Time (h)	5.512	6.029	6.541	6.861
Cost/animation (Thousands of dollars)	4.032	4.410	4.785	5.019
Performance/Pixel (ms)	11.880	12.993	14.098	14.787

The average duration of a movie or animation is 106 minutes at 24 fps, for a total of $24 \times 106 \times 60 = 152640$ images [9]. The animation cost is calculated using the rendering time, hardware power consumption, and according to [86], the U.S. cost per kilowatt-hour is 10.65 cents. Equation 6.1 shows the formula used.

$$Cost/animation = \frac{Watts \ \times \ Hours - Used}{1000} \ \times \ Cost \ per \ kilowatt - hour \ \times \ 152640 \tag{6.1}$$

Performance/Pixel was calculated by averaging the number of pixels in the three resolutions used for the experiment. The average time to render an image was divided by the average number of pixels in an image.

6.5.1 Specific scenario

As [16] express, a real-world rendering scenario involves high resolutions and a combination of images' effects. These are the combinations where rendering took the most time in all the OS/scheduling policy combinations. The scenarios are:

1. Table 6.8 shows the metrics obtained for the case where all the scenes were at

higher resolution (i.e., 1280x720). A combination with a lower value indicates a better performance.

2. Table 6.9 shows the metrics obtained for the case where 'Contemporary Bathroom' was selected at higher resolution (i.e., 1280x720). This scenario was selected because it was the one that took the most rendering time. As in the previous, combination with lower value has better performance.

Table 6.8: Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP at higher resolution (lower is better).

Metric	SPOS:CSP	SPOS:CFSSP	GPOS:CSP	GPOS:CFSSP
Average Rendering Time (h)	9.137	10.152	10.905	11.416
Cost/animation (Thousands of dollars)	6.684	7.426	7.977	8.351
Performance/Pixel (ms)	19.692	21.879	23.502	24.603

Table 6.9: Obtained metrics for the SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP at higher resolution and most complex scene (lower is better).

Metric	SPOS:CSP	SPOS:CFSSP	GPOS:CSP	GPOS:CFSSP
Average Rendering Time (h)	36.619	39.751	42.829	43.676
Cost/animation (Thousands of dollars)	26.788	29.079	31.331	31.950
Performance/Pixel (ms)	78.921	85.671	92.305	94.129

6.5.2 SPOS with scheduling policy comparison

Table 6.10 shows the overall performance improvement obtained using SPOS with the custom scheduling policy against the other combinations for the general case and the two specific cases. A higher number is better for the SPOS with the custom scheduling policy.

Table 6.10: Performance improvement of the SPOS:CSP against SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP.

Scenario	SPOS:CFSSP	GPOS:CSP	GPOS:CFSSP
General case (%)	8.568	15.732	19.659
Specific scenario 1 (%)	9.998	16.211	19.963
Specific scenario 2 (%)	7.878	14.499	16.157

6.5.3 Actual rendering time reduction

Table 6.11 shows the actual rendering time obtained in the experiment. It shows that the experiment was performed for almost one hundred and thirty-one days. Using SPOS with the custom scheduling policy shows a clear reduction in the rendering time greater than seven days.

Table 6.11: Actual rendering time reduction

OS	Scheduling Policy	Rendering Time (h)	Rendering Time (days)
GPOS	CFS Scheduling Policy	864.50099722	36.020874884
GPOS	Custom Scheduling Policy	824.21819722	34.342424884
SPOS	CFS Scheduling Policy	759.63733611	31.651555671
SPOS	Custom Scheduling Policy	694.54930556	28.939554398
Total		3142.906	130.954

6.6 Correctness

The correctness of the proposed implementation was tested to verify that all the four implementations (i.e., SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP) render the same image under the same conditions. The output images were visually validated, but a formal test was performed using the `diff` command of the GPOS. It compares two files, and if there is no difference between them, the option `-s` will display a message saying that 'the files are identical'.

The test was performed for scenes rendered under the same conditions (scene name and resolution) but with different OS/scheduling policy conditions. Figure 6.19 shows an example of the analysis, and as expected, the command's output says that the images are identical.

```
alvaro@alvaro-GL62-6QF:~$ diff -s GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr GPOS_Breakfast_1280x720_CSP_afinity_iter_0.exr
Files GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr and GPOS_Breakfast_1280x720_CSP_afinity_iter_0.exr are identical
alvaro@alvaro-GL62-6QF:~$ diff -s GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr SPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr
Files GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr and SPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr are identical
alvaro@alvaro-GL62-6QF:~$ diff -s GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr GPOS_Breakfast_1280x720_CSP_afinity_iter_0.exr
Files GPOS_Breakfast_1280x720_CFSSP_afinity_iter_0.exr and GPOS_Breakfast_1280x720_CSP_afinity_iter_0.exr are identical
alvaro@alvaro-GL62-6QF:~$
```

Figure 6.19: Correctness check using output images.

(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

Figure 6.20: Barcelona Pavilion scene at resolution of 1280x720

6.7 Output images

Figure 6.20 to Figure 6.26 show scenes at 1280x720 resolution for the four OS/scheduling policy (i.e., SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP). They are the actual output from the experiments.

(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

Figure 6.21: Bathroom scene at resolution of 1280x720

(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

Figure 6.22: Breakfast scene at resolution of 1280x720

(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

Figure 6.23: Contemporary Bathroom scene at resolution of 1280x720

(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

Figure 6.24: Crown scene at resolution of 1280x720

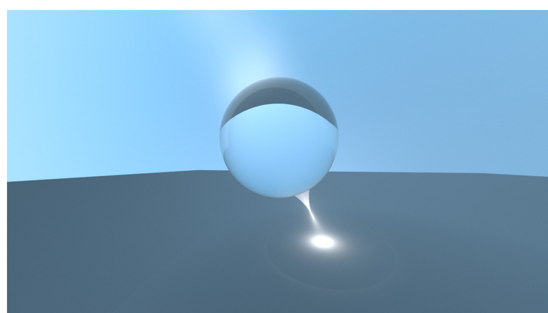
(a) GPOS with custom scheduling policy

(b) GPOS with CFS scheduling policy

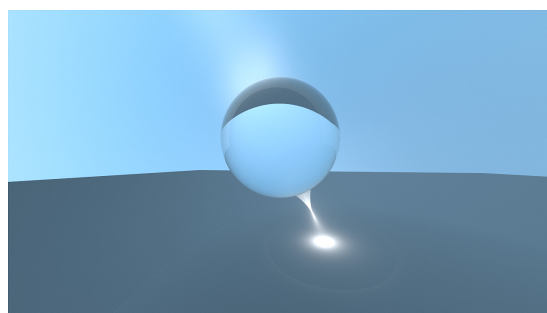
(c) SPOS with custom scheduling policy

(d) SPOS with CFS scheduling policy

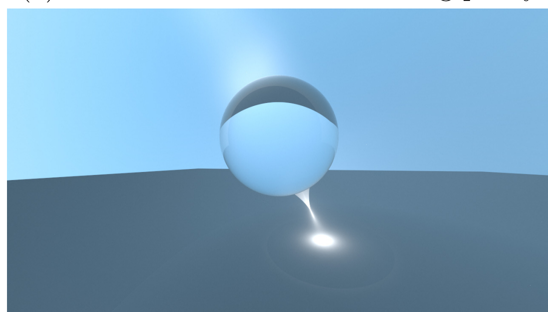
Figure 6.25: Landscape scene at resolution of 1280x720



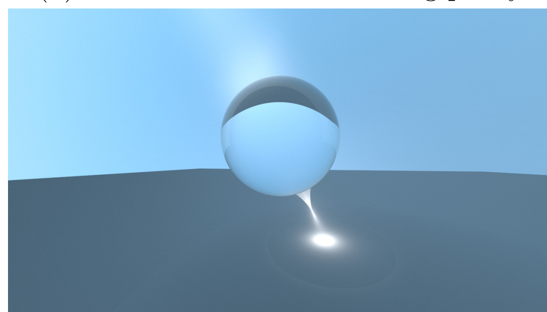
(a) GPOS with custom scheduling policy



(b) GPOS with CFS scheduling policy



(c) SPOS with custom scheduling policy



(d) SPOS with CFS scheduling policy

Figure 6.26: Volume Caustic scene at resolution of 1280x720

6.8 Other results

There are some results also significant to mention as part of the outputs of this research.

- **A functional recipe for implementing PBRT-v3 as a layer of the Yocto Project**

  As mentioned, PBRT-v3 works as the library to execute the image rendering using the ray tracing algorithm. No documentation related to PBRT-v3 implemented as part of the Yocto Project was found during the development of the SPOS. A new functional recipe was designed as part of the implementation of the SPOS. This recipe will be delivered to the community. Appendix D shows the details of this recipe. Figure 6.27 shows a screenshot of PBRT-v3 up and running in the SPOS.

Figure 6.27: System running SPOS with PBRT-v3 in a x86 architecture

- **A guide with the most relevant steps to implement a scheduling policy in a Linux based system**

  The Design Chapter described the main files and code segments that must be modified and added to implement a custom scheduling policy. This research's development and implementation time was long due to the lack of information on implementing a new scheduling policy in a modern Linux's kernel. It may work as a baseline for future research efforts to avoid spend several months digging in the

Linux's kernel source code to understand the kernel's scheduling code and interconnection.

Chapter 7

Discussion

This chapter presents the analysis and discussion of the results presented in chapter Results.

Analyzing the results from Table 6.1 is clear that the use of the custom scheduling policy has a positive impact on the reduction of cache misses, context switches, and thread migration. Firstly, we can see that the hypothesis was true by confirming that setting hard cache affinity reduces the number of cache misses. The reduction of cache misses compared against baseline was 16% which is a significant improvement. Secondly, the context switch is a painful process for the system because it must refresh the cache, implying the loss of the 'warm cache' condition. The reduction of context switches was up to 32%. The migration statistic can explain the performance gain. As we can see, in the case of the custom scheduling policy, there is no migration between cores because the thread is only allowed to run in a specific CPU. It implies that the custom scheduling policy the rendering time by having a warmer cache (numerous cache hits) through the hard cache affinity.

A preliminary observation of the ray tracing algorithm response was performed. Analyzing the behavior of the scheduling policy in Figure 6.1 seems that when custom scheduling policy is assigned to a PBRT task, it produces an apparent reduction of the rendering time. Another preliminary result is shown in Figure 6.2, where using a SPOS

seems another factor that improves the system's performance when executing the ray tracing algorithm. More in-depth analysis done, including other factors, is presented in Figure 6.3 and Figure 6.4. It shows how the combination of SPOS and custom scheduling policy is a combination that reduces the rendering time when multiple changes in the environment, such as image resolutions and scene complicity, are added to the system. Again, these results are preliminary and considered anecdotes until the formal ANOVA analysis was conducted with the data gathered from the different factors and levels defined in the Methodology.

Because the original data did not satisfy the ANOVA assumptions, the ANOVA analysis was performed with the transformed response variable to comply with normality and equality of variance (homoscedasticity) assumptions. Figure 6.14 presents the results of the ANOVA test. Analyzing the F-Ratio column demonstrates that all the factors and their combinations influence the response variable (rendering time). Resolution impacts the rendering time on how many pixels the engine must process in the image, and this implies a ray for that pixel, intersection detection, and effects calculation. All of this directly impacts rendering time as the image resolution is present in the complexity of the ray tracing algorithm as the I factor in the $O(I\, log\, n)$ rendering time [16]. The scene's name is the name of the factor that sweeps the real-world scenes. It impacts the performance directly because in the ray-tracing algorithms, for each ray, its intersection against each object in the scene must be calculated. The more objects and complexity in the scene, the more intersections must be detected, generating more memory access and mathematical operations. Here is where the addition of hard cache affinity (using the custom scheduling policy) impacts the gain in performance because the number of cache misses is reduced due to reducing context switches in the processor.

The most critical factors to test the proposed research hypothesis are the OS and custom scheduling policy. Table 6.3 demonstrates that the mean rendering time for the combination SPOS:CSP against the combinations SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP are statistically different. Figure 6.15 shows the mean rendering time for

each combination and confirms that SPOS:CSP produces the maximum performance.

A more in-depth analysis was performed to investigate the rendering time in the resolution and OS/custom scheduling policy combination. Mean rendering time for each resolution must contain all the possible combinations of scenes. Table 6.4 shows that the mean time of the combination SPOS:CSP against SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP is statistically different when comparing the exact image resolution. Figure 6.16 helps to demonstrate that SPOS:CSP has the best performance when a different resolution is applied to the image. Another critical point is that the resolution impacts workload directly because each increment in the resolution implies other rays and an additional intersection calculation. The figure shows that SPOS:CSP is potent at higher resolutions because the gaps between the other combinations are more significant at maximum resolution.

Table 6.5 displays the data when a scene is changed. The SPOS:CSP's rendering time (again) is statistically different compared to SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP when a scene changes in the experiment. Figure 6.17 is the mean plot that confirms that SPOS:CSP has a better performance than SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP when a scene changes in the experiment. It is clear that SPOS:CSP, no matter the scene in the experiment, has a lower rendering time. Another relevant detail is that for all the scenes where SPOS is used, the rendering time is lower than GPOS. It can be explained by reminding the critical element of a SPOS, a specialized operating system designed to execute one or just a few tasks compared to GPOS. In this case, the change in the scene implies a modification in the image complexity, which is translated to an increment in the mathematical operation that requires more memory access. Setting hard cache affinity through the custom scheduling policy prevents the cache misses that implies that when the ray tracing algorithm is being executed, the data required will be in the cache most of the time (warm cache).

Table 6.6 display the data when all the factors are interacting with each other. As expected, the rendering time for the ray tracing algorithm for the SPOS:CSP is sta-

tistically different to SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP when all the factors interact. Figure 6.18 shows the mean plot for this interaction. It is evident that SPOS:CSP produces the best performance across all factors' interaction when compared against SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP. Another point to mention from this figure is that it is easy to observe the rendering time increment due to the image resolution. In addition, from this table we can observed that the proposed solution has a maximum rendering time reduction of 23% approximately.

Several metrics were obtained for the proposed mechanism to compare the performance of the SPOS:CSP against the SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP, as shown in Table 6.7. From this table it can be inferred three aspects:

1. The SPOS:CSP is the software infrastructure with the lowest average rendering time for all the combinations of scenarios defined in the experiment, which means it has the best performance.
2. The SPOS:CSP has the lowest cost based on rendering time and power consumption.
3. The SPOS:CSP is the combination with a lower rendering time per pixel than the other combinations.

These results undoubtedly confirm that SPOS:CSP is the best software infrastructure for rendering animations as it delivers the best performance at a lower cost.

Table 6.8 and Table 6.9 show the metrics for two specific cases: 1) Scenes with the higher resolution and 2) Scene Contemporary Bathroom with higher resolution. Again, the combination of SPOS:CSP is better in average rendering time, cost per animation, and performance per pixel for combining factors that closely approach a real-world rendering scenario. Another aspect confirms that SPOS:CSP holds the performance advantage even in cases that require the most computation power, which provides an insight that this software implementation has the potential for accelerating computationally intensive workloads.

Table 6.10 allows assessing whether the proposed hypothesis for this research effort is correct. The combination of SPOS:CSP demonstrates a gain of more than 10% performance in the three scenarios against the baseline combination (GPOS:CFSSP), confirming that the hypothesis was correct. When SPOS:CSP is compared against GPOS:CSP, it also has more than 10% performance gain. Finally, when SPOS:CSP is compared against SPOS:CFSSP, it has an average performance gain of 8.8% in the three scenarios analyzed, which is a clear indication of the power of the proposed custom scheduling policy.

Table 6.11 shows that the total rendering time of the experiment performed in this research work lasted approximately 3143 hours for the 504 tests. It is equivalent to almost 131 days of continuous rendering. The baseline took 36 days, while the experiment with the proposed configuration took approximately 29 days, that is, seven days less. It represents a solution with great potential to reduce the time-to-market of audiovisual productions such as those mentioned at the beginning of this presentation. It demonstrates the success of our solution and highlights the complexity and demand for computational resources by the rendering process.

A correctness test was performed, checking that the output images are the same for each OS/scheduling policy combination. Figure 6.19 shows that output images are the same through the command diff. This test is essential to validate that the ANOVA test was an "apples-to-apples" comparison.

Several examples of rendered images using the ray-tracing algorithm are shown from Figure 6.20 to Figure 6.26. The images are at maximum resolution for each OS/scheduling policy combination (i.e., SPOS:CSP, SPOS:CFSSP, GPOS:CSP, and GPOS:CFSSP).

The recipe for implementing PBRT-v3 for the SPOS meets the software requirements to execute the ray tracing algorithm in the SPOS. PBRT-v3 was successfully implemented for the first time in an OS based on Yocto Project. The steps followed to implement the new scheduling policy were also successfully proved. This scheduling policy was used by PBRT tasks to assign the hard cache affinity.

Chapter 8

Conclusions and Future Work

This chapter presents the conclusions obtained through this research effort and the future work that can be achieved to improve the results and/or try another approach.

8.1 Conclusions

This research aimed to reduce the rendering time through the design and the implementation of a custom scheduling policy in a SPOS. Based on a statistical analysis of different rendering scenarios, it can be concluded that the combination of SPOS with the custom scheduling policy reduces the rendering time by 19% and showing a maximum reduction of 23% compared to the baseline GPOS with the CFS scheduling policy. The results indicate the potential to reduce the time-to-market because the proposal showed to be the most cost-effective software platform for rendering ray-traced images through a lower rendering time and performance per pixel. Furthermore, the hard cache affinity demonstrates to reduce the cache misses by 16% and context switching by 32%. Indeed, this solution accelerates computationally intensive workload, such as ray tracing without any alteration to the output image and regardless of image resolution and scene complexity. Finally, this research demonstrates that creating a new scheduling policy for Linux's kernel can be an excellent method to accelerate computationally intensive workloads.

8.2 Future Work

Future work concerns deeper analysis of particular mechanisms, new proposals to try different methods, or simply curiosity. This research was mainly focused on the assignation of hard cache affinity as a method to reduce the rendering time of ray tracing through the design and implementation of a custom scheduling policy in a SPOS, leaving a few studies outside the scope of the thesis. An in-deep profiling effort to explore any other method to accelerate ray tracing in the operating system. Other libraries to execute ray tracing or implement the proposed solution in an optimized OS, such as RTOS, can be future research efforts. In addition, using powerful and modern hardware can boost up the results of this research. Along with other methods to implement hard cache affinity in a Linux-based operating system, a deeper analysis of all the non-analyzed ANOVA factor interactions can provide more insights into the behavior of ray tracing in the environment proposed. Furthermore, the use of other scenes and image resolutions can explore the behavior of the proposed solution in more complex scenarios. Finally, an in-deep analysis helps to understand the main contributors that produce such a difference in the performance between a SPOS and a GPOS.

Bibliography

[1] Abraham Silberschatz, Galvin Peter Baer, and Greg Gagne. *Operating System Concepts*. 2018.

[2] Gautam Bhanage. What datastructure does the CFS use and why, 2 2017.

[3] Edward R. Freniere and John Tourtellott. Brief history of generalized ray tracing. In *Lens Design, Illumination, and Optomechanical Modeling*, volume 3130, pages 170–178. SPIE, 9 1997.

[4] Curtis Vermeeren. Raytracer in C++, 10 2016.

[5] Steven G. Parker, Heiko Friedrich, David Luebke, Keith Morley, James Bigler, Jared Hoberock, David McAllister, Austin Robison, Andreas Dietrich, Greg Humphreys, Morgan McGuire, Martin Stich, and et al. GPU Ray Tracing. *Commun. ACM*, 56(5):93–101, 5 2013.

[6] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Scenes for pbrt-v3.

[7] Sanford E. DeVoe and Jeffrey Pfeffer. When time is money: The effect of hourly payment on the evaluation of time. *Organizational Behavior and Human Decision Processes*, 104(1):1–13, 9 2007.

[8] Angelo Dalle Molle. 3 Ways to Reduce Time to Market in Your Company | AlfaPeople-Global, 10 2018.

[9] Przemysław Jarząbek. Are new movies longer than they were 10, 20, 50 year ago? | by Przemysław Jarząbek | Towards Data Science, 12 2018.

[10] Per H Christensen, Julian Fong, David M Laur, and Dana Batali. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[11] Kyle Desiderio and Ian Phillips. How Pixar's animation has evolved over 24 years, from 'Toy Story' to 'Toy Story 4', 6 2019.

[12] Peter Collingridge. Ask a Pixar Scientist | The Science Behind Pixar.

[13] Steve Rotenberg. Rendering Algorithms, 2017.

[14] John Hart, Nathan A Carr, Jesse D Hall, and John C Hart. Chapter 8. Rendering Algorithms The Ray Engine. Technical report, 2002.

[15] Physically Based Rendering. In Matt Pharr, Wenzel Jakob, and Greg Humphreys, editors, *Physically Based Rendering (Third Edition)*, page iv. Morgan Kaufmann, Boston, third edit edition, 2017.

[16] Ernesto Rivera-Alvarado and Francisco Torres-Rojas. APU performance evaluation for accelerating computationally expensive workloads. 2 2019.

[17] Ernesto Rivera-Alvarado. *APU performance evaluation for accelerating computationally expensive workloads*. PhD thesis, ITCR, 2019.

[18] Alexandru Voica. Ray tracing for beginners, 2015.

[19] David Cardinal. How Nvidia's RTX Real-Time Ray Tracing Works - ExtremeTech, 2018.

[20] Daniel Etiemble. 45-year CPU evolution: one law and two equations. *Second Workshop on Pioneering Processor Paradigms*, 2018.

[21] Alex Glawion. Best CPU For Rendering [2020 Guide], 5 2020.

[22] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems - SIGMETRICS '93*, pages 272–274, New York, New York, USA, 1993. ACM Press.

[23] Vahid Kazempour, Alexandra Fedorova, Pouya Alagheband, and E Luque. Performance implications of cache affinity on multicore processors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5168 LNCS:151–161, 2008.

[24] Matthias Diener, Eduardo H.M. Cruz, Marco A.Z. Alves, Philippe O.A. Navaux, Anselm Busse, and Hans Ulrich Heiss. Kernel-Based Thread and Data Mapping for Improved Memory Affinity. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2653–2666, 9 2016.

[25] José Jaime Ruz Ortiz. Tema 6: Memoria Caché, 2013.

[26] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design & Implementation 3rd Edition*. 2006.

[27] Ali Mohammed, Ahmed Eleliemy, Florina M Ciorba, Franziska Kasielke, and Ioana Banicescu. An Approach for Realistically Simulating the Performance of Scientific Applications on High Performance Computing Systems. Technical report, 10 2019.

[28] Kernel.org. CFS Scheduler.

[29] Samih M Mostafa, Hirofumi Amano, and Shigeru Kusakabe. Fairness and High Performance for Tasks in General Purspose Multicore Systems. 29(December):74–86, 2016.

[30] Jean Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: A decade of wasted cores. *Proceedings of the 11th European Conference on Computer Systems, EuroSys 2016*, 2016.

[31] Sunita Dhotre, Pooja Patil, S. H. Patil, and Rucha Jamale. Analysis of scheduler settings on the performance of multi-core processors. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, volume 2018-Janua, pages 687–691, 5 2017.

[32] Peter Kogge, Study Lead, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely Thomas, Sterling R Stanley, and Williams Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, 2008.

[33] L Jochen and J Liedtke. On Micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 237–250, New York, NY, USA, 1995. Association for Computing Machinery.

[34] Dawson R Engler, M Frans Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95*, pages 251–266, New York, New York, USA, 1995. ACM Press.

[35] Jon K. Carrol. How To: Building Your Own Render Farm | Tom's Hardware, 3 2010.

[36] iNET PC. Best 3D rendering computer: Waukesha techs help Milwaukee designers | iNET PC Waukesha, Wisconsin.

[37] William George. What is the Best CPU for Rendering (2019), 12 2019.

[38] Rai Technology University. Advanced Computer Architecture. 2016.

[39] Douglas Comer. *Essentials of Computer Architecture, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2017.

[40] David A Patterson and John L Hennessy. In Praise of Computer Organization and Design: The Hardware/ Software Interface, Fifth Edition. Technical report, 2014.

[41] Matt Safforf. How to Buy the Right CPU: A Guide for 2020, 2 2020.

[42] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. Suggested ways to teach the art of multiprocessor programming. *The Art of Multiprocessor Programming*, pages xxi–xxii, 1 2021.

[43] Peter YK Cheung. Introduction to Memories and Computer Architecture, 2018.

[44] Sandro Matheus V.N. Marques, Thiarles S. Medeiros, Fabio D. Rossi, Marcelo C. Luizelli, Alessandro G. Girardi, Antonio Carlos S. Beck, and Arthur F. Lorenzon. The Impact of Turbo Frequency on the Energy, Performance, and Aging of Parallel Applications. In *IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*, volume 2019-Octob, pages 149–154. IEEE Computer Society, 10 2019.

[45] Bo Moore. CPU Temperature Overheat | PC Gamer, 9 2016.

[46] Nikita Ishkov and Martti Juhola. A complete guide to Linux process scheduling. Technical report, 2015.

[47] Chris A. MacK. Fifty years of Moore's law. In *IEEE Transactions on Semiconductor Manufacturing*, volume 24, pages 202–207, 5 2011.

[48] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., 1973.

[49] Intel Corp. Improving Real-Time Performance by Utilizing Cache Allocation Technology Enhancing Performance via Allocation of the Processor's Cache White Paper. Technical report, 2015.

[50] Hyeonjoong Cho, E Douglas Jensen, Binoy Ravindran, and E Douglas Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, page 101–110, USA, 2006. IEEE Computer Society.

[51] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux J.*, 2009(184), 8 2009.

[52] Ludwig Hellgren Winblad. CFS (Completely Fair Scheduler) in the Linux kernel.

[53] Daniel Kopta. Ray Tracing from a Data Movement Perspective. (May), 2016.

[54] Arthur Appel. Some techniques for shading machine renderings of solids, 1968.

[55] Robert Glanz. A Comparison of Physically Based Rendering Systems, 2018.

[56] David R Cheriton and Kenneth J Duda. A Caching Model of Operating System Kernel Functionality. *SIGOPS Oper. Syst. Rev.*, 29(1):83–86, 1 1995.

[57] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Process Scheduling Challenges in the Era of Multi-Core Processors. *Intel Technology Journal*, 11(4), 2007.

[58] Alexandros Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios Koziris. LCA: A memory link and cache-aware co-scheduling approach for CMPs. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 469–470, 2014.

[59] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 11 1988.

[60] Joost Van Pinxten, Marc Geilen, and Twan Basten. Parametric Scheduler Characterization. *ACM Trans. Embed. Comput. Syst.*, 18(5s), 10 2019.

[61] R T Gollapudi, G Yuksek, and K Ghose. Cache-Aware Dynamic Classification and Scheduling for Linux. In *2019 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pages 1–3, 4 2019.

[62] W Li, S Mohanty, and K Kavi. A Page-based Hybrid (Software-Hardware) Dynamic Memory Allocator. *IEEE Computer Architecture Letters*, 5(2):13, 2 2006.

[63] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse, and Hans Ulrich Heiß. kMAF: Automatic kernel-level management of thread and data affinity. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 277–288, 2014.

[64] Kinson Chan, King Tin Lam, and Cho Li Wang. Cache affinity optimization techniques for scaling software transactional memory systems on multi-CMP architectures. *Proceedings - IEEE 14th International Symposium on Parallel and Distributed Computing, ISPDC 2015*, pages 56–65, 7 2015.

[65] PassMark Software. PassMark CPU Benchmarks - Year on Year Performance, 2020.

[66] Sherri L. Smith. Best gaming laptops of 2020: Top gaming laptops ranked | Laptop Mag, 8 2020.

[67] Jorge Jimenez. Best gaming laptops for 2020 | PC Gamer, 8 2020.

[68] Barbara G Tabachnick and Linda S Fidell. *Experimental Designs Using ANOVA*. 2007.

[69] B. L. Agarwal, editor. *Basic Statistics* . New Ager International Ltd., Delhi, fourth edition edition, 2006.

[70] Ke Gu, Min Liu, Guangtao Zhai, Xiaokang Yang, and Wenjun Zhang. Quality Assessment Considering Viewing Distance and Image Resolution. *IEEE Transactions on Broadcasting*, 61(3):520–531, 9 2015.

[71] Intel Corp. Intel® Core™ i7-6700HQ Processor (6M Cache, up to 3.50 GHz) Product Specifications, 2015.

[72] Michael Kerrisk. taskset(1) - Linux manual page, 8 2020.

[73] Yocto Project. About – Yocto Project.

[74] Sarah Boslaugh and Paul Andrew Watters. *Statistics in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, first edit edition, 7 2008.

[75] S. J. Amster. Beyond ANOVA, Basics of Applied Statistics. *Technometrics*, 29(3):387–387, 8 1987.

[76] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Tutorial - Perf Wiki, 2015.

[77] Andy P. Field. *Discovering statistics using SPSS : (and sex and drugs and rock 'n' roll).* SAGE Publications, 2009.

[78] Marius Marusteri and Vladimir Bacarea. Comparing groups for statistical differences: how to choose the right statistical test? *Biochemia Medica*, 20(1):15–32, 2 2010.

[79] Michael H. Herzog, Gregory Francis, and Aaron Clarke. ANOVA. pages 67–82, 2019.

[80] Salvatore S Mangiafico. *SUMMARY AND ANALYSIS OF EXTENSION PROGRAM EVALUATION IN R, version 1.18.8.* 2016.

[81] UC Business Analytics R Programming Guide. Assessing the Assumptions of Homogeneity · UC Business Analytics R Programming Guide.

[82] Tom Donnelly. Transforming Data to Make Better Predictions , 3 2021.

[83] Rob J Hyndman and George Athanasopoulos. Forecasting: Principles and Practice (2nd ed), 2018.

[84] Jason T. Newsom. Post Hoc Tests. *Univariate Quantitative Methods*, 2020.

[85] Neil Salkind. Tukey's Honestly Significant Difference (HSD). *Encyclopedia of Research Design*, 10 2012.

[86] U.S. Energy Information Administration. Electric Power Monthly - U.S. Energy Information Administration (EIA), 5 2021.

Appendix A

Implemented source code

Many changes and source-code addition was required to implement the custom scheduling policy. Here is the ordered list of the most relevant files modified in this research to add the new scheduling policy named **custom_scheduler**.

- ***$Kernel_ source-code*/arch/x86/kconfig**

  Since x86 is the host system's architecture, the first step is to add a new configuration entry in *kconfig*. Listing A.1 shows the new configuration entry. This entry will be referend to as CONFIG_SCHED_CUSTOM_POLICY, but here the prefix CONFIG_ is omitted.

  Listing A.1: kconfig configuration option entry

  ```
  menu "Custom scheduler"
  config SCHED_CUSTOM_POLICY
      bool "Custom scheduling policy"
      default y
  endmemu

  ```

- ***$Kernel_ source-code*/include/uapi/linux/sched.h**

  The macro to identify the new scheduling policy must be added in this file. Listing A.2 shows the implementation.

Listing A.2: Scheduling policy macro definition

```
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
.
.
.
#ifdef CONFIG_SCHED_CUSTOM_POLICY
#define SCHED_CUSTOM_POLICY     7
#endif

```

- ***$Kernel_ source-code*/include/linux/custom/custom.h**

  Listing A.3 shows the header file implemented. The definition of *struct list_ head* is required to organized the tasks in a doubled linked list. The field *custom_ root* in the *custom_ rq* data struct is used to specify the root of the PBRT task on the run queue. The *spinlock_ t* variable is defined to protect list operation. Finally, the *init_ custom_ rq* initializes the *struct custom_ rq*.

Listing A.3: Code inside custom.h header.

```
#ifndef __CUSTOM_H_
#define __CUSTOM_H_

#include <linux/sched.h>
#include <linux/list.h>
#include <linux/spinlock.h>

struct custom_rq{
    struct list_head custom_list;
    struct rb_root custom_root;
    spinlock_t lock;
};
void init_custom(struct custom_rq *rq);
#endif
```

```
    };

```

- ***$Kernel_ source-code*/kernel/sched/sched.h**

  struct rq is a data structure that holds a run-queue of all runnable processes assigned to it. The scheduling policies use this run queue to select the best process to be executed when the current process is done. Finally, the struct *custom_rq* is added to the struct rq. Listing A.4 shows the implementation.

Listing A.4: Adding data struct to manage PBRT tasks

```
    #ifdef CONFIG_SCHED_CUSTOM_POLICY
    #include <linux/custom/custom.h>
    #endif

    struct rq {
    ...
        #ifdef CONFIG_SCHED_CUSTOM_POLICY
        struct custom_rq custom_rq;
        #endif
    ...
    };

```

- ***$Kernel_ source-code*/kernel/sched/custom/custom_rq.c**

  struct custom_rq must be initialized before the kernel scheduler starts operation. *init_custom_rq* function was implemented to initialize all the fields in the data structure, as shown in Listing A.5.

Listing A.5: Initializing custom_rq data structure

```
    #include <linux/custom_rq.h>

    void init_rbtree(rbtree **t) {
        *t=malloc(sizeof(rbtree));
```

```
5
6         if (*t==NULL) {
7             //handle error
8         }
9         else {
10            (*t)->root = NULL;
11
12            node *n = malloc(sizeof(node));
13            if(n != NULL){
14                n->color = BLACK;
15                (*t)->nil = n;
16            }
17        }
18    }
19
20    void init_custom_rq(struct custom_rq *rq)
21    {
22        INIT_LIST_HEAD(&rq->custom_list);
23        spin_lock_init(&rq->lock);
24        init_rbtree(&rq->custom_root);
25    }
26
27    };
28
```

- ***$Kernel_ source-code*/kernel/sched/core.c**

  The data structure described above must be initialized. Listing A.6 shows the modifications in function *sched_init* to initialize the data structure. *sched_init* function creates the CPU run queues.

Listing A.6: Adding data struct to manage PBRT tasks

```
1    void __init sched_init(void){
2    ...
3        for_each_possible_cpu(i){
```

```
4               ...
5       #ifdef CONFIG_SCHED_CUSTOM_POLICY
6               init_custom_rq(&rq-->custom_rq);
7       #endif
8               ...
9           }
10      ...
11      }
12
```

- ***$Kernel_ source-code*/include/linux/custom/custom_task.h**

  "custom_task" is another data structure added. This structure will store the information for each task that uses the custom scheduling policy. As was mentioned above, CFS is the baseline for this implementation. The data type *struct rb_node* refers to the rbtree used by CFS to organize the tasks. Listing A.7 shows the implementation.

Listing A.7: Creating custom_task.h header.

```
1   #ifndef __CUSTOM_TASK_H_
2   #define __CUSTOM_TASK_H_
3
4   #include <linux/types.h>
5
6   struct custom_task{
7       struct rb_node custom_node;
8       struct list_head custom_list_node;
9       bool is_pbrt;
10      ...
11  };
12  };
13
```

- ***$Kernel_ source-code*/include/linux/sched.h**

This file includes one critical data structure: *struct task_struct*. *struct task_struct* is known as the process descriptor. The kernel uses this structure to maintain information about each process. Some of the parameters stored here are run-state, priority, scheduling class, address space, and affinity mask. task_struct is stored in a circular doubly-linked list. Listing A.8 shows the changes at the structure *task_struct*. *struct custom_task* was added to contain the data for the custom task.

Listing A.8: Changes at *struct task_struct*

```
#ifdef CONFIG_SCHED_CUSTOM_POLICY
#include <linux/custom/custom_task.h>
#endif

...

struct task_struct {
...
#ifdef CONFIG_SCHED_CUSTOM_POLICY
    struct custom_task custom_task;
#endif
}

```

- ***$Kernel_ source-code*/kernel/sched/custom/custom_sched_policy.c**

  Each native scheduling policy such as RT (real-time policy), CFS, or IDLE has its file that describes its behavior. As mentioned above, the custom scheduling policy is based on CFS but includes specific changes that differentiate from the native CFS. Listing A.9 shows a set of functions that are mandatory for each policy. **next** is a pointer to the following scheduling policy in the priority hierarchy. As defined, the custom scheduling policy will have a higher priority than the original RT scheduling policy. The change in the priority was necessary because, in a few experiments,

PBRT tasks used other scheduling policies instead of the custom scheduling policy. Other functions are showed. These auxiliary callback functions react to certain events like adding or removing tasks from the linked list.

Listing A.9: Adding custom scheduling policy source code

```
1  const struct sched_class custom_sched_class = {
2   .next   = &rt_sched_class,
3   .enqueue_task  = enqueue_task_custom,
4   .dequeue_task  = dequeue_task_custom,
5   .yield_task  = yield_task_custom,
6
7   .check_preempt_curr = check_preempt_curr_custom,
8
9   .pick_next_task  = pick_next_task_custom,
10  .put_prev_task  = put_prev_task_custom,
11 #ifdef CONFIG_SMP
12     .select_task_rq = select_task_rq_custom,
13 #endif
14
15
16     .set_curr_task      = set_next_task_custom,
17  .task_tick      = task_tick_custom,
18
19  .get_rr_interval = get_rr_interval_custom,
20
21  .prio_changed  = prio_changed_custom,
22  .switched_to  = switched_to_custom,
23
24  .update_curr  = update_curr_custom,
25 ...
26 };
27
```

- ***$Kernel_source-code*/kernel/sched/custom/custom_sched_policy.c**

The custom scheduling policy is only applied to the PBRT tasks. When the task is created, it receives a name stored into **comm**, a member of the *task_struct*. The custom scheduling policy will know if the task is a PBRT task to continue with the hard cache affinity. Listing A.10 shows the implementation of the function. If the task is named '*pbrt*', it implies that the custom scheduling policy must change the parameter *is_pbrt* to '1' (true). If the task is not a PBRT process, it means that the custom scheduling policy must modify the current scheduling policy based on its priority.

Listing A.10: Function to identify if the task is a PBRT task

```
...
#include <linux/types.h>
#include <linux/string.h>
#include <linux/sched.h>
...

void is_pbrt_task(struct task_struct *p)
{
    char custom_task [TASK_COMM_LEN];
    struct sched_param aux_param = { .sched_priority = 0 };

    strcpy(custom_task, "pbrt");

    if (strcmp(custom_task,p->comm))
    {
        p->custom_task->is_pbrt = 1;
        p->prio = 0;
    }
    else
    {
        if (task_has_rt_policy(p))
        {
      aux_param.sched_priority = p->rt_priority;
```

```
24            sched_setscheduler(&p, SCHED_FIFO, &aux_param);
25        }
26        else
27        {
28            sched_setscheduler(&p, SCHED_NORMAL, &aux_param);
29        }
30          }
31      }
32      ...
33
```

Listing A.11 shows another critical function implemented as part of the custom scheduling policy. *cpu_custom_mask_set* is the function that provides the algorithm to assign hard cache affinity to each PBRT task. An assumption made in this implementation is that the processor has eight cores.

Listing A.11: Function to set cache affinity in the new custom scheduling policy

```
1     ...
2     #include <linux/pid.h>
3     #include <linux/sched/types.h>
4     #include <linux/cpumask.h>
5     #include <linux/sched.h>
6     ...
7
8     #define thread1_bitmap 0x01
9     #define thread2_bitmap 0x02
10    #define thread3_bitmap 0x04
11    #define thread4_bitmap 0x08
12    #define thread5_bitmap 0x010
13    #define thread6_bitmap 0x020
14    #define thread7_bitmap 0x040
15    #define thread8_bitmap 0x080
16    ...
17
```

```
18  void cpu_custom_mask_set(struct task_struct *p)
19  {
20      struct cpumask_t *cpumask_thread;
21      unsigned int CPUS_AV;
22      int i = 1;
23      pid_t parent_task_pdi;
24      struct list_head task_threads, *aux_list_head, *threads;
25  
26      cpumask_thread = to_cpumask(thread1_bitmap);
27      parent_task_pdi = p->pid;
28      sched_setaffinity(parent_task_pdi, *cpumask_thread1);
29  
30      task_threads = p->children;
31  
32      CPUS_AV = num_online_cpus();
33  
34      list_for_head(aux_list_head, task_threads);
35      {
36          if ((i <= CPUS_AV) &&  (i == 1)){
37              cpumask_thread = to_cpumask(thread1_bitmap);
38          }
39          else if ((i <= CPUS_AV) &&  (i == 2)){
40              cpumask_thread = to_cpumask(thread2_bitmap);
41          }
42          else if ((i <= CPUS_AV) &&  (i == 3)){
43              cpumask_thread = to_cpumask(thread3_bitmap);
44          }
45          else if ((i <= CPUS_AV) &&  (i == 4)){
46              cpumask_thread = to_cpumask(thread4_bitmap);
47          }
48          else if ((i <= CPUS_AV) &&  (i == 5)){
49              cpumask_thread = to_cpumask(thread5_bitmap);
50          }
51          else if ((i <= CPUS_AV) &&  (i == 6)){
```

```
52                    cpumask_thread = to_cpumask(thread6_bitmap);
53                }
54                else if ((i <= CPUS_AV) &&  (i == 7)){
55                    cpumask_thread = to_cpumask(thread7_bitmap);
56                }
57                else if ((i <= CPUS_AV) &&  (i == 8)){
58                    cpumask_thread = to_cpumask(thread8_bitmap);
59                }
60                threads = list_entry(aux_list_head, struct task_struct, p
    );
61                sched_setaffinity(threads->pid, *cpumask_thread);
62            }
63
64
65     }
66     ...
67
```

The function enqueue_task_custom described in Listing A.12 is called whenever a PBRT task (custom_task) enters a runnable state.

Listing A.12: Function to add custom task (PBRT) to the run queue

```
1      ...
2      /* Walk up scheduling entities hierarchy */
3      #define for_each_sched_entity(se) \
4    for (; se; se = se->parent)
5   ...
6      static void __enqueue_entity(struct custom_rq *custom_rq, struct
     sched_entity *se)
7      {
8       struct rb_node **link = &custom_rq->rb_root.rb_node;
9       struct rb_node *parent = NULL;
10      struct sched_entity *entry;
11      bool leftmost = true;
12
```

```
13      /*
14       * Find the right place in the rbtree:
15       */
16      while (*link) {
17       parent = *link;
18       entry = rb_entry(parent, struct sched_entity, run_node);
19       /*
20        * We dont care about collisions. Nodes with
21        * the same key stay together.
22        */
23       if (entity_before(se, entry)) {
24        link = &parent->rb_left;
25       } else {
26        link = &parent->rb_right;
27        leftmost = false;
28       }
29      }
30
31      rb_link_node(&se->run_node, parent, link);
32      rb_insert_color_cached(&se->run_node,&cfs_rq->tasks_timeline,
    leftmost);
33     }
34
35     static void enqueue_task_custom(struct rq *rq, struct task_struct
     *p, int flags)
36     {
37         is_pbrt_task(&p);
38         if (p->custom_task->is_pbrt == 1)
39             break;
40         for_each_sched_entity(se){
41             if (p->se->on_rq)
42        break;
43             spin_lock(&rq->custom_rq.lock);
44             cpu_custom_mask_set(&p);
```

```
45              list_add(&p->custom_task.custom_list_node,&rq->custom_rq.
    custom_list);
46              __enqueue_entity(&rq->custom_rq, &p->se)
47              spin_unlock(&rq->custom_rq.lock);
48          }
49      }
50
```

Similar to *enqueue_ task_ custom* described above, the function to dequeue a custom task was declared. Listing A.13 shows this function required when a custom (PBRT) task is no longer runnable. Even if the tasks were not executed, they might be removed from the double-linked list.

Listing A.13: Function to remove custom task (PBRT) to the run queue

```
1   static void
2   static void __dequeue_entity(struct custom_rq *custom_rq, struct
    sched_entity *se)
3   {
4    rb_erase_cached(&se->run_node, &custom_rq->custom_root);
5   }
6
7   dequeue_entity(struct custom_rq *custom_rq, struct sched_entity *
    se, int flags)
8   {
9
10   update_curr(custom_rq);
11
12   update_load_avg(custom_rq, se, UPDATE_TG);
13   dequeue_runnable_load_avg(custom_rq, se);
14
15   update_stats_dequeue(custom_rq, se, flags);
16
17   clear_buddies(custom_rq, se);
18
```

```
19        if (se != custom_rq->curr)
20         __dequeue_entity(custom_rq, se);
21        se->on_rq = 0;
22        account_entity_dequeue(custom_rq, se);
23
24        if (!(flags & DEQUEUE_SLEEP))
25         se->vruntime -= custom_rq->min_vruntime;
26
27        /* return excess runtime on last dequeue */
28        return_cfs_rq_runtime(custom_rq);
29
30        update_customs_group(se);
31
32
33        if ((flags & (DEQUEUE_SAVE | DEQUEUE_MOVE)) != DEQUEUE_SAVE)
34         update_min_vruntime(custom_rq);
35      }
36
37      static void deenqueue_task_custom(struct rq *rq, struct
       task_struct *p, int flags)
38      {
39          struct lf_custom *t = NULL;
40          spin_lock(&rq->custom_rq.lock);
41          list_del(&p->custom_task.custom_list_node);
42          if(list_empty(&rq->custom_rq.custom_list))
43          {
44              rq->&rq->custom_rq.custom_list = NULL;
45          }
46          else
47          {
48              t = list_first_entry(&rq->custom_rq.custom_list,struct
       custom_task, custom_list_node);
49              for_each_sched_entity(p->se)
50              {
```

```
51                  dequeue_entity(cfs_rq, se, flags);
52              }
53
54          }
55          spin_unlock(&rq->lf.lock);
56      }
57
```

- ***$Kernel_ source-code*/kernel/sched/sched/sched.h**

One important assumption made in this research project is that the custom scheduling policy implemented has the highest priority. This assumption is essential to ensure the PBRT tasks run using the developed custom scheduling policy. During the testing process was detected some scenarios where PBRT tasks used another scheduling policy. Listing A.14 shows the changes in some functions in the *sched.h* header.

Listing A.14: Adding Custom Policy to the priority hierarchy

```
1     #ifdef CONFIG_SCHED_CUSTOM_POLICY
2     extern const struct sched_class custom_sched_class;
3     #endif
4     ...
5     #ifdef CONFIG_SCHED_CUSTOM_POLICY
6     static inline int custom_policy(int policy)
7     {
8         return policy = SCHED_CUSTOM_POLICY;
9  #endif
10    }
11    ...
12    static inline bool valid_policy(int policy)
13    {
14    #ifdef CONFIG_SCHED_CUSTOM_POLICY
15     return idle_policy(policy) || fair_policy(policy) ||
16      rt_policy(policy) || dl_policy(policy) || custom_policy(policy)
```

```
 ;
17 #else
18     return idle_policy(policy) || fair_policy(policy) ||
19   rt_policy(policy) || dl_policy(policy);
20 #endif
21 }
22
23 static inline int task_has_custom_policy(struct task_struct *p)
24 {
25  return custom_policy(p->policy);
26 }
27 ...
28
```

- ***$Kernel_ source-code*/kernel/sched/sched/core.h**

  The changes described by Listing A.15 are required to assign the scheduling class to the *task_ struct*.

Listing A.15: Changing *setscheduling()* function

```
1 ...
2 static void __setscheduler(struct rq *rq, struct task_struct *p,
3   const struct sched_attr *attr, bool keep_boost)
4 {
5  ...
6     if (p->policy == SCHED_CUSTOM_POLICY)
7         p->sched_class = &custom_sched_class;
8     else if (dl_prio(p->prio))
9   p->sched_class = &dl_sched_class;
10  else if (rt_prio(p->prio))
11   p->sched_class = &rt_sched_class;
12  else
13   p->sched_class = &fair_sched_class;
14 }
15 ...
```

Appendix B

Gathered Data from Experiments

This section presents the data obtained from the experiments that were used to perform the ANOVA test. Each row represents a case and the respective rendering time for the image that was generated. In Table B.1 the first column, “Scene’s Name”, specifies the scene’s name. The second column, “OS”, indicates the operating system used in the rendering. The third column, “Scheduling Policy”, indicates whether Custom Scheduling Policy (CSP) or CFS Scheduling Policy (CFSSP) was set. The fourth column, “Resolution’, indicates the resolution of the output image. The fifth column, “Iteration”, indicates the number of iteration for each case. The sixth column, “Rendering Time (s)”, contains the rendering time in seconds. Finally, column “Rendering Time (h)” indicates the rendering time in hours.

Table B.1: Data obtained from the experiments

Observation	Scene's Name	OS	Scheduling Policy	Resolution	Rendering Time (h)
1	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.9269444444
2	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.9177777778
3	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.9261111111
4	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.9236111111
5	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.9313888889
6	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	1280x720	3.925
7	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	1.0063888889
8	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	0.9944444444
9	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	1.0033333333
10	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	0.9961111111
11	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	1.0008333333
12	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	640x360	0.9969444444
13	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.2202777778
14	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.2069444444
15	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.22
16	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.2111111111

17	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.21
18	Barcelona_Pavilion	GPOS	Custom Scheduling Policy	960x540	2.2091666667
19	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9688888889
20	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9730555556
21	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9627777778
22	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9563888889
23	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9536111111
24	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	1280x720	3.9786111111
25	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0077777778
26	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0036111111
27	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0066666667
28	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0011111111
29	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0038888889
30	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	640x360	1.0041666667
31	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2452777778
32	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2497222222
33	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2563888889
34	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2544444444

35	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2472222222
36	Barcelona_Pavilion	GPOS	CFS Scheduling Policy	960x540	2.2411111111
37	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.5972222222
38	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.6005555556
39	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.6022222222
40	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.5822222222
41	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.5827777778
42	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	1280x720	3.6058333333
43	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.9152777778
44	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.9133333333
45	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.9169444444
46	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.9161111111
47	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.9136111111
48	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	640x360	0.915
49	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.0663888889
50	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.0641666667
51	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.0691666667
52	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.0725

53	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.07
54	Barcelona_Pavilion	SPOS	Custom Scheduling Policy	960x540	2.0580555556
55	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0147222222
56	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0105555556
57	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0163888889
58	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0222222222
59	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0075
60	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	1280x720	4.0183333333
61	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.9802777778
62	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.9841666667
63	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.985
64	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.9825
65	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.9838888889
66	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	640x360	0.9783333333
67	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.1736111111
68	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.18
69	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.1744444444
70	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.1794444444

71	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.1702777778
72	Barcelona_Pavilion	SPOS	CFS Scheduling Policy	960x540	2.1858333333
73	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.3277777778
74	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.3113888889
75	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.3361111111
76	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.2930555556
77	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.285
78	Bathroom	GPOS	Custom Scheduling Policy	1280x720	8.2694444444
79	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.3041666667
80	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.3013888889
81	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.3094444444
82	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.3044444444
83	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.2930555556
84	Bathroom	GPOS	Custom Scheduling Policy	640x360	2.2963888889
85	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.6738888889
86	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.6694444444
87	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.6480555556
88	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.6430555556

89	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.64
90	Bathroom	GPOS	Custom Scheduling Policy	960x540	4.6280555556
91	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.1641666667
92	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.1905555556
93	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.2019444444
94	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.2063888889
95	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.1841666667
96	Bathroom	GPOS	CFS Scheduling Policy	1280x720	9.1713888889
97	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.6933333333
98	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.6958333333
99	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.6891666667
100	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.6875
101	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.7019444444
102	Bathroom	GPOS	CFS Scheduling Policy	640x360	2.6961111111
103	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.16
104	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.1586111111
105	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.1738888889
106	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.1722222222

107	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.1555555556
108	Bathroom	GPOS	CFS Scheduling Policy	960x540	5.1463888889
109	Bathroom	SPOS	Custom Scheduling Policy	1280x720	6.9463888889
110	Bathroom	SPOS	Custom Scheduling Policy	1280x720	6.9908333333
111	Bathroom	SPOS	Custom Scheduling Policy	1280x720	6.9825
112	Bathroom	SPOS	Custom Scheduling Policy	1280x720	7.0125
113	Bathroom	SPOS	Custom Scheduling Policy	1280x720	6.9847222222
114	Bathroom	SPOS	Custom Scheduling Policy	1280x720	6.9561111111
115	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.8913888889
116	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.8988888889
117	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.8911111111
118	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.9019444444
119	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.9011111111
120	Bathroom	SPOS	Custom Scheduling Policy	640x360	1.8977777778
121	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9566666667
122	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9777777778
123	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9788888889
124	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9744444444

125	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9580555556
126	Bathroom	SPOS	Custom Scheduling Policy	960x540	3.9558333333
127	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.7044444444
128	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.7119444444
129	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.7080555556
130	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.7311111111
131	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.7119444444
132	Bathroom	SPOS	CFS Scheduling Policy	1280x720	7.6738888889
133	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.3013888889
134	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.3083333333
135	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.3161111111
136	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.2966666667
137	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.3166666667
138	Bathroom	SPOS	CFS Scheduling Policy	640x360	2.2963888889
139	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2361111111
140	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2316666667
141	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2294444444
142	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2447222222

143	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2219444444
144	Bathroom	SPOS	CFS Scheduling Policy	960x540	4.2322222222
145	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.8127777778
146	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.7847222222
147	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.8208333333
148	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.7997222222
149	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.8197222222
150	Breakfast	GPOS	Custom Scheduling Policy	1280x720	4.8244444444
151	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.2375
152	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.2413888889
153	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.2358333333
154	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.2377777778
155	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.2352777778
156	Breakfast	GPOS	Custom Scheduling Policy	640x360	1.235
157	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.6933333333
158	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.6819444444
159	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.69
160	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.6891666667

161	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.6808333333
162	Breakfast	GPOS	Custom Scheduling Policy	960x540	2.6880555556
163	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9444444444
164	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9447222222
165	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9591666667
166	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9419444444
167	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9608333333
168	Breakfast	GPOS	CFS Scheduling Policy	1280x720	4.9719444444
169	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3611111111
170	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3544444444
171	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3536111111
172	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3594444444
173	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3513888889
174	Breakfast	GPOS	CFS Scheduling Policy	640x360	1.3591666667
175	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7777777778
176	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7722222222
177	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7886111111
178	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7702777778

179	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7677777778
180	Breakfast	GPOS	CFS Scheduling Policy	960x540	2.7680555556
181	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.1013888889
182	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.0933333333
183	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.0808333333
184	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.1130555556
185	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.0930555556
186	Breakfast	SPOS	Custom Scheduling Policy	1280x720	4.1069444444
187	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0266666667
188	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0377777778
189	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0386111111
190	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0316666667
191	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0372222222
192	Breakfast	SPOS	Custom Scheduling Policy	640x360	1.0366666667
193	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3416666667
194	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3338888889
195	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3283333333
196	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3225

197	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3241666667
198	Breakfast	SPOS	Custom Scheduling Policy	960x540	2.3175
199	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9305555556
200	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9333333333
201	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9405555556
202	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9438888889
203	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9122222222
204	Breakfast	SPOS	CFS Scheduling Policy	1280x720	4.9522222222
205	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2330555556
206	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2372222222
207	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2402777778
208	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2394444444
209	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2377777778
210	Breakfast	SPOS	CFS Scheduling Policy	640x360	1.2425
211	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.6891666667
212	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.6933333333
213	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.7019444444
214	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.6830555556

215	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.6833333333
216	Breakfast	SPOS	CFS Scheduling Policy	960x540	2.6841666667
217	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	42.763333333
218	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	42.814722222
219	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	42.917222222
220	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	43.037222222
221	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	42.66
222	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	1280x720	42.783611111
223	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.729444444
224	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.685833333
225	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.726388889
226	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.741388889
227	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.691944444
228	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	640x360	10.752777778
229	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.082222222
230	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.075277778
231	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.078611111
232	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.078611111

233	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.1025
234	Contemporary_Bathroom	GPOS	Custom Scheduling Policy	960x540	24.008611111
235	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.606388889
236	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.643888889
237	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.702222222
238	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.568888889
239	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.714722222
240	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	1280x720	43.82
241	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.917222222
242	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.904722222
243	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.925833333
244	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.916111111
245	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.916111111
246	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	640x360	10.970555556
247	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.506666667
248	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.535833333
249	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.536111111
250	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.560833333

251	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.608888889
252	Contemporary_Bathroom	GPOS	CFS Scheduling Policy	960x540	24.518333333
253	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.611944444
254	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.763333333
255	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.614166667
256	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.455277778
257	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.721944444
258	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	1280x720	36.550277778
259	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0569444444
260	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0263888889
261	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0052777778
262	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0369444444
263	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0041666667
264	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	640x360	9.0080555556
265	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.224166667
266	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.185
267	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.143888889
268	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.103888889

269	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.224722222
270	Contemporary_Bathroom	SPOS	Custom Scheduling Policy	960x540	20.261388889
271	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.666666667
272	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.825277778
273	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.666388889
274	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.811944444
275	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.939444444
276	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	1280x720	39.597777778
277	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.6077777778
278	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.5102777778
279	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.5780555556
280	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.5811111111
281	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.5455555556
282	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	640x360	9.6188888889
283	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.177222222
284	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.1775
285	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.283055556
286	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.131388889

287	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.142222222
288	Contemporary_Bathroom	SPOS	CFS Scheduling Policy	960x540	21.221944444
289	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2719444444
290	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2769444444
291	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2719444444
292	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2883333333
293	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2811111111
294	Crown	GPOS	Custom Scheduling Policy	1280x720	5.2722222222
295	Crown	GPOS	Custom Scheduling Policy	640x360	1.3283333333
296	Crown	GPOS	Custom Scheduling Policy	640x360	1.3283333333
297	Crown	GPOS	Custom Scheduling Policy	640x360	1.3255555556
298	Crown	GPOS	Custom Scheduling Policy	640x360	1.3236111111
299	Crown	GPOS	Custom Scheduling Policy	640x360	1.3277777778
300	Crown	GPOS	Custom Scheduling Policy	640x360	1.3244444444
301	Crown	GPOS	Custom Scheduling Policy	960x540	2.99
302	Crown	GPOS	Custom Scheduling Policy	960x540	2.99
303	Crown	GPOS	Custom Scheduling Policy	960x540	2.975
304	Crown	GPOS	Custom Scheduling Policy	960x540	2.9875

305	Crown	GPOS	Custom Scheduling Policy	960x540	2.9722222222
306	Crown	GPOS	Custom Scheduling Policy	960x540	2.9941666667
307	Crown	GPOS	CFS Scheduling Policy	1280x720	5.7091666667
308	Crown	GPOS	CFS Scheduling Policy	1280x720	5.6897222222
309	Crown	GPOS	CFS Scheduling Policy	1280x720	5.7163888889
310	Crown	GPOS	CFS Scheduling Policy	1280x720	5.7013888889
311	Crown	GPOS	CFS Scheduling Policy	1280x720	5.7047222222
312	Crown	GPOS	CFS Scheduling Policy	1280x720	5.6758333333
313	Crown	GPOS	CFS Scheduling Policy	640x360	1.4380555556
314	Crown	GPOS	CFS Scheduling Policy	640x360	1.4383333333
315	Crown	GPOS	CFS Scheduling Policy	640x360	1.4258333333
316	Crown	GPOS	CFS Scheduling Policy	640x360	1.4283333333
317	Crown	GPOS	CFS Scheduling Policy	640x360	1.4352777778
318	Crown	GPOS	CFS Scheduling Policy	640x360	1.4330555556
319	Crown	GPOS	CFS Scheduling Policy	960x540	3.1841666667
320	Crown	GPOS	CFS Scheduling Policy	960x540	3.1966666667
321	Crown	GPOS	CFS Scheduling Policy	960x540	3.2116666667
322	Crown	GPOS	CFS Scheduling Policy	960x540	3.1994444444

323	Crown	GPOS	CFS Scheduling Policy	960x540	3.2063888889
324	Crown	GPOS	CFS Scheduling Policy	960x540	3.1994444444
325	Crown	SPOS	Custom Scheduling Policy	1280x720	4.39
326	Crown	SPOS	Custom Scheduling Policy	1280x720	4.39
327	Crown	SPOS	Custom Scheduling Policy	1280x720	4.3677777778
328	Crown	SPOS	Custom Scheduling Policy	1280x720	4.3775
329	Crown	SPOS	Custom Scheduling Policy	1280x720	4.4044444444
330	Crown	SPOS	Custom Scheduling Policy	1280x720	4.3747222222
331	Crown	SPOS	Custom Scheduling Policy	640x360	1.1188888889
332	Crown	SPOS	Custom Scheduling Policy	640x360	1.1261111111
333	Crown	SPOS	Custom Scheduling Policy	640x360	1.1266666667
334	Crown	SPOS	Custom Scheduling Policy	640x360	1.1202777778
335	Crown	SPOS	Custom Scheduling Policy	640x360	1.1252777778
336	Crown	SPOS	Custom Scheduling Policy	640x360	1.1230555556
337	Crown	SPOS	Custom Scheduling Policy	960x540	2.4744444444
338	Crown	SPOS	Custom Scheduling Policy	960x540	2.4766666667
339	Crown	SPOS	Custom Scheduling Policy	960x540	2.4827777778
340	Crown	SPOS	Custom Scheduling Policy	960x540	2.4647222222

341	Crown	SPOS	Custom Scheduling Policy	960x540	2.4675
342	Crown	SPOS	Custom Scheduling Policy	960x540	2.4638888889
343	Crown	SPOS	CFS Scheduling Policy	1280x720	4.8769444444
344	Crown	SPOS	CFS Scheduling Policy	1280x720	4.8866666667
345	Crown	SPOS	CFS Scheduling Policy	1280x720	4.8622222222
346	Crown	SPOS	CFS Scheduling Policy	1280x720	4.9002777778
347	Crown	SPOS	CFS Scheduling Policy	1280x720	4.8619444444
348	Crown	SPOS	CFS Scheduling Policy	1280x720	4.8575
349	Crown	SPOS	CFS Scheduling Policy	640x360	1.2077777778
350	Crown	SPOS	CFS Scheduling Policy	640x360	1.2138888889
351	Crown	SPOS	CFS Scheduling Policy	640x360	1.2119444444
352	Crown	SPOS	CFS Scheduling Policy	640x360	1.2052777778
353	Crown	SPOS	CFS Scheduling Policy	640x360	1.2158333333
354	Crown	SPOS	CFS Scheduling Policy	640x360	1.2052777778
355	Crown	SPOS	CFS Scheduling Policy	960x540	2.8283333333
356	Crown	SPOS	CFS Scheduling Policy	960x540	2.8422222222
357	Crown	SPOS	CFS Scheduling Policy	960x540	2.8266666667
358	Crown	SPOS	CFS Scheduling Policy	960x540	2.8444444444

359	Crown	SPOS	CFS Scheduling Policy	960x540	2.8408333333
360	Crown	SPOS	CFS Scheduling Policy	960x540	2.8386111111
361	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.5013888889
362	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.4947222222
363	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.4913888889
364	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.4755555556
365	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.5177777778
366	Landscape	GPOS	Custom Scheduling Policy	1280x720	6.4758333333
367	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3563888889
368	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3630555556
369	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3555555556
370	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3516666667
371	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3547222222
372	Landscape	GPOS	Custom Scheduling Policy	640x360	1.3544444444
373	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5119444444
374	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5216666667
375	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5188888889
376	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5216666667

377	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5230555556
378	Landscape	GPOS	Custom Scheduling Policy	960x540	2.5194444444
379	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2725
380	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2577777778
381	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2358333333
382	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2275
383	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2475
384	Landscape	GPOS	CFS Scheduling Policy	1280x720	7.2622222222
385	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5530555556
386	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5575
387	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5552777778
388	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5552777778
389	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5552777778
390	Landscape	GPOS	CFS Scheduling Policy	640x360	1.5552777778
391	Landscape	GPOS	CFS Scheduling Policy	960x540	2.8761111111
392	Landscape	GPOS	CFS Scheduling Policy	960x540	2.8616666667
393	Landscape	GPOS	CFS Scheduling Policy	960x540	2.8688888889
394	Landscape	GPOS	CFS Scheduling Policy	960x540	2.8658333333

395	Landscape	GPOS	CFS Scheduling Policy	960x540	2.8647222222
396	Landscape	GPOS	CFS Scheduling Policy	960x540	2.855
397	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.4227777778
398	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.43
399	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.4286111111
400	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.4272222222
401	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.4275
402	Landscape	SPOS	Custom Scheduling Policy	1280x720	4.4138888889
403	Landscape	SPOS	Custom Scheduling Policy	640x360	1.1038888889
404	Landscape	SPOS	Custom Scheduling Policy	640x360	1.1016666667
405	Landscape	SPOS	Custom Scheduling Policy	640x360	1.1019444444
406	Landscape	SPOS	Custom Scheduling Policy	640x360	1.0977777778
407	Landscape	SPOS	Custom Scheduling Policy	640x360	1.1058333333
408	Landscape	SPOS	Custom Scheduling Policy	640x360	1.1041666667
409	Landscape	SPOS	Custom Scheduling Policy	960x540	2.4002777778
410	Landscape	SPOS	Custom Scheduling Policy	960x540	2.3975
411	Landscape	SPOS	Custom Scheduling Policy	960x540	2.4058333333
412	Landscape	SPOS	Custom Scheduling Policy	960x540	2.3933333333

413	Landscape	SPOS	Custom Scheduling Policy	960x540	2.4086111111
414	Landscape	SPOS	Custom Scheduling Policy	960x540	2.4102777778
415	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7405555556
416	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7116666667
417	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7316666667
418	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7125
419	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7058333333
420	Landscape	SPOS	CFS Scheduling Policy	1280x720	5.7219444444
421	Landscape	SPOS	CFS Scheduling Policy	640x360	1.1438888889
422	Landscape	SPOS	CFS Scheduling Policy	640x360	1.1463888889
423	Landscape	SPOS	CFS Scheduling Policy	640x360	1.1466666667
424	Landscape	SPOS	CFS Scheduling Policy	640x360	1.1452777778
425	Landscape	SPOS	CFS Scheduling Policy	640x360	1.1436111111
426	Landscape	SPOS	CFS Scheduling Policy	640x360	1.145
427	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4555555556
428	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4530555556
429	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4544444444
430	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4494444444

431	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4558333333
432	Landscape	SPOS	CFS Scheduling Policy	960x540	2.4538888889
433	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.7006944444
434	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.6977777778
435	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.7006944444
436	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.6953472222
437	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.6938888889
438	Volume_Caustic	GPOS	Custom Scheduling Policy	1280x720	4.6788194444
439	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.2745444444
440	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.2783555556
441	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.277675
442	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.2746805556
443	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.2749527778
444	Volume_Caustic	GPOS	Custom Scheduling Policy	640x360	1.2691
445	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.6802777778
446	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.6983333333
447	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.6944444444
448	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.6925

449	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.695
450	Volume_Caustic	GPOS	Custom Scheduling Policy	960x540	2.7038888889
451	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.18
452	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.15375
453	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.1717361111
454	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.1853472222
455	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.1945833333
456	Volume_Caustic	GPOS	CFS Scheduling Policy	1280x720	5.1936111111
457	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3250416667
458	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3194611111
459	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3247694444
460	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3172833333
461	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3191888889
462	Volume_Caustic	GPOS	CFS Scheduling Policy	640x360	1.3215027778
463	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.0769444444
464	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.0861111111
465	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.0813888889
466	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.0936111111

467	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.0902777778
468	Volume_Caustic	GPOS	CFS Scheduling Policy	960x540	3.1011111111
469	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8577777778
470	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8514583333
471	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8640972222
472	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8689583333
473	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8606944444
474	Volume_Caustic	SPOS	Custom Scheduling Policy	1280x720	3.8451388889
475	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0465583333
476	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0586722222
477	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0514583333
478	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0475111111
479	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0471027778
480	Volume_Caustic	SPOS	Custom Scheduling Policy	640x360	1.0540444444
481	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2233333333
482	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2305555556
483	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2205555556
484	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2327777778

485	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2333333333
486	Volume_Caustic	SPOS	Custom Scheduling Policy	960x540	2.2180555556
487	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.0497916667
488	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.0609722222
489	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.055625
490	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.0716666667
491	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.0706944444
492	Volume_Caustic	SPOS	CFS Scheduling Policy	1280x720	4.0531944444
493	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.1457833333
494	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.1425166667
495	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.1463277778
496	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.14415
497	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.1482333333
498	Volume_Caustic	SPOS	CFS Scheduling Policy	640x360	1.147825
499	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.3608333333
500	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.3661111111
501	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.3622222222
502	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.36

503	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.3641666667
504	Volume_Caustic	SPOS	CFS Scheduling Policy	960x540	2.3622222222

Appendix C

Compiling a custom kernel for Ubuntu OS

This section presents the step-by-step guide to compile a custom kernel for Ubuntu OS. This guide was followed to create the general-purpose operating system with the custom scheduling policy.

1. Install dependencies. The packages required to build the kernel image are listed below.

```
1  sudo apt-get install git build-essential kernel-package fakeroot
   libncurses5-dev libssl-dev ccache bison flex libelf-dev dwarves
2
```

2. Create a folder to build the kernel. For example $HOME/KernelExample

3. Move to KernelExample:
   cd KernelExample

4. Clone the kernel from Ubuntu's git:

```
1  git clone -b linux-5.4.y git://git.kernel.org/pub/scm/linux/
   kernel/git/stable/linux-stable.git
2
```

5. Move to linux-stable:

   cd linux-stable

6. Add the source code required to the kernel. For this research, the files added and modified are described in the Design.

7. Copy the configuration from the existing kernel into it.

```
cp /boot/config-`uname -r` .config

```

8. In most cases, the configuration should be changed to generate a custom kernel for the GPOS. The configuration was left as in the original kernel to generate the same conditions as the original GPOS. In case the configuration must be changed, it can be accessed through make menuconfig. Figure C.1 shows the kernel configuration menu.

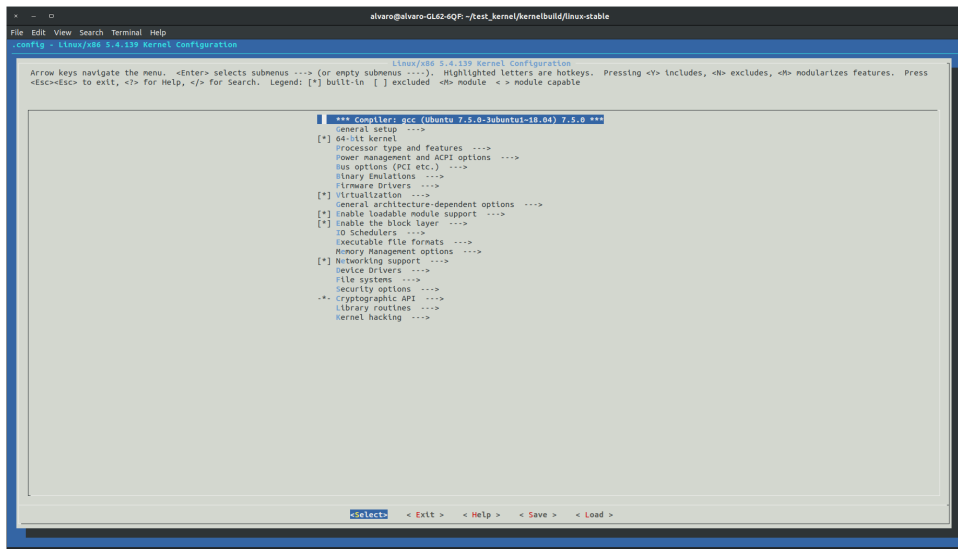


Figure C.1: Kernel configuration.

Once the configuration is done, save the configuration and exit.

9. Then, clean the directory.

```
make clean

```

10. The next step is to build the kernel. It may take hours to compile the kernel.

```
make -j8 LOCALVERSION=-customGPOS

```

The option '-j8' allows using the eight cores of the host's architecture. The number after the j would depend on the host's cores. The option *LOCALVERSION* add 'customGPOS' to the kernel's name.

11. Finally, the created kernel should be installed through the dpkg command.

```
cd ..
sudo dpkg -i linux-firmware-image_5.4.139-customGPOS-1_amd64.deb
sudo dpkg -i linux-libc-dev_5.4.139-customGPOS-1_amd64.deb
sudo dpkg -i linux-headers-_5.4.139-customGPOS-1_amd64.deb
sudo dpkg -i linux-image-dbg_5.4.139-customGPOS-1_amd64.deb
sudo dpkg -i linux-image-_5.4.139-customGPOS-1_amd64.deb

```

When the installation finishes, restart the host. The default kernel should be the one just built.

Appendix D

Recipe to include PBRT-v3 in the SPOS based on Yocto Project

D.1 Yocto Recipe to Implement

This section will explain the steps to include PBRT-v3 in custom OS based on Yocto Project. This section does not guide the reader on the step-by-step procedure to set up the whole environment to get the SPOS. This section only focuses on the recipe to compile PBRT-v3. For further information to set up Yocto's environment, the following links can be accessed:

Yocto Project's Reference Manual: https://www.yoctoproject.org/docs/2.5/ref-manual/ref-manual.html

'Hello World' with Yocto Project:

1. https://variwiki.com/index.php?title=Yocto_Hello_World
2. https://wiki.yoctoproject.org/wiki/Building_your_own_recipes_from_first_principles

D.1.1 Prerequisites to build PBRT from source code

PBRT-v3 is the library adopted to render the scenes used in the experiments of this thesis. PBRT-v3 is licensed under the **BSD 2-Clause "Simplified" License**. It means that it can be used and modified without breaking any license right.

The source code can be downloaded from here: https://github.com/mmp/pbrt-v3. PBRT's website (https://www.pbrt.org) can be accessed for further information about the library.

Two important disclaimers:

1. This procedure is applicable for x86 architecture. Likely, following the following steps will work on other architectures like ARM.
2. Only PBRT-v3 was used in this work. It means that previous versions may not work following the steps described in this appendix.

Prerequisites steps:

1. **Download PBRT-v3** from https://github.com/mmp/pbrt-v3
2. **Get the compiler for the cross-compilation**. The current method implies cross-compilation for some files that PBRT requires for its compilation. The files are:
   - **toFloat.cpp**: It will create the header *toFloat.h*.

     cpp location: ./PBRT-v3/src/ext/openexr/IlmBase/Half/

     Where to save toFloat.h? ./PBRT-v3/src/ext/openexr/IlmBase/Half/
   - **half.cpp**: It will create the header *half.h*.

     cpp location: ./PBRT-v3/src/ext/openexr/IlmBase/Half/

     Where to save half.h? ./PBRT-v3/src/ext/openexr/IlmBase/Half/
   - **eLut.cpp**: It will create the header *eLut.h*.

cpp location: ./PBRT-v3/src/ext/openexr/IlmBase/Half/

Where to save eLut.h? ./PBRT-v3/src/ext/openexr/IlmBase/Half/

- **dwaLookups.cpp**: It will create the header *dwaLookups.h*.

  cpp location: ./PBRT-v3/src/ext/openexr/OpenEXR/Ilmlmf/

  Where to save dwaLookups.h? ./PBRT-v3/src/ext/openexr/OpenEXR/Ilmlmf/

The host uses x86 architecture. The compiler used to compile the above files was the same compiler as the host. A ready-to-go version of PBRT-v3 for Yocto for x86 architecture can be downloaded at: https://bitbucket.org/acm_0993/thesis/raw/69168b320af528bf15389beedca7547d3f013e57/pbrt-v3_1.1.tar.gz

3. **Set the Yocto environment**. For this research effort, the release **YP Core - Dunfell 3.1.4 - 2020.12.02** was used. It can be cloned by copying/pasting in a terminal the following command:

```
git clone -b dunfell git://git.yoctoproject.org/poky.git

```

4. **Dependencies:** The dependencies to compile PBRT-v3 are listed out below:

   - **Doxygen**: It can be added to the compilation by following the recipe from the official Yocto Project's repository: http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/tree/recipes-devtools/doxygen/doxygen_1.8.9.1.bb?h=thud.

     zlib: To meet the dependency would be enough to add *DEPENDS = "zlib"* in PBRT-v3's recipe, as explained in the next section.

D.1.2 Creating the Yocto's Recipe for PBRT-v3

The Yocto Project's layer model is an efficient way to build a custom Linux distribution by including/removing proprietary modules. It allows the user to share, collaborate, reuse,

change instructions and settings at any time, separate information in a custom build.

For this implementation, the meta layer that includes PBRT-v3's recipe was called *meta-pbrt*. Inside *meta-pbrt*, a folder named *recipes-pbrt-v3* contains the recipe to compile PBRT-v3 using the custom version created by following the prerequisites listed in the above section.

The recipe for PBRT-v3 follows the standard Yocto Project's recipe structure. The recipe is shown in Listing D.1. An important detail is that the recipe uses the ready-to-go implementation used in the thesis.

Listing D.1: PBRT-v3 Recipe for Yocto

```
1  SUMMARY =  "PBRT v3 compilation"
2  SECTION = "PBRT"
3  LICENSE = "MIT"
4  LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835
       ade698e0bcf8506ecda2f7b4f302"
5
6  FILESEXTRAPATHS_prepend := "${THISDIR}/${BPN}_${PV}:"
7
8  SRC_URI = "https://bitbucket.org/acm_0993/thesis/raw/69168
       b320af528bf15389beedca7547d3f013e57/${BPN}_${PV}.tar.gz"
9  SRC_URI[sha256sum] = "
       ccc0fba85c43123c50864e066a948d94a61d85231f0209e44fe550a14fd069a8"
10
11 S = "${WORKDIR}"
12
13 DEPENDS = "zlib"
14
15 inherit cmake
```

The recipe uses an MIT license. The checksum may change in a custom implementation. Another detail is that the recipe includes *zlib* as a PBRT-v3 dependency, as mentioned above.

Once the recipe and Yocto's environment are ready, the next step would be building the custom Linux image containing PBRT-v3. For testing purposes, a *minimal-image* with QEMU support is recommended.