# Composed task graph's dynamic scheduling

Baptiste Coye



THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

# DOCTEUR
# DE L'UNIVERSITÉ DE BORDEAUX

Spécialité informatique

ECOLE DOCTORALE MATHEMATIQUE ET

INFORMATIQUE

Par **Baptiste COYE**

## Ordonnancement dynamique de graphe de tâches par composition

Sous la direction de : **Denis BARTHOU**
Co-dirigée par : **Raymond NAMYST**

Soutenue le 10 Juillet 2023

Membres du jury :

| | | | |
|---|---|---|---|
| M. Tristan CAZENAVE | Professeur des Universités | Université Paris Dauphine | Rapporteur |
| M. William JALBY | Professeur des Universités | Université Versaille Saint-Quentin | Rapporteur |
| Mme. Elisabeth BRUNET | Maître de conférence | Telecom Sud Paris | Examinatrice |
| M. Laurent SIMON | Professeur des Universités | Bordeaux INP | Examinateur |
| M. Hervé HUBELE | Lead Programmer | Ubisoft | Invité |
| M. Denis BARTHOU | Professeur des Universités | Bordeaux INP | Directeur |
| M. Raymond NAMYST | Professeur des Universités | Université de Bordeaux | Directeur |

# Contents

# Acknowledgments

Reaching the end of this Ph.D journey has proven to be a deeply impactfull and transformative experience. Over the course of these three years, I've learned as much about myself as I have about my chosen field of study. This has not only enriched my professional growth but has also been instrumental in my personal evolution and I am deeply grateful to the many individuals who have supported me along the way.

First and foremost, I'd like to express my sincere gratitude to my three supervisors, Denis, Hervé and Raymond for your unwavering support have been instrumental in shaping my research, nurturing my thoughts and . I am honored to have had the privilege of learning from you.

My deepest thanks also go to my colleagues from Ubisoft and fellow researchers at Inria, your camaraderie, and intellectual discussions have enriched my research and made this journey all the more enjoyable. Citing all of you would be too long, but please know that each coffee break (or afternoon) was a genuine pleasure.

To my family, friends, your encouragement, patience, alcoholism, and belief in me have been my pillars of strength throughout this endeavor. I am profoundly thankful for your constant support and understanding.

Lastly, to Charline, I know no words are required as none would suffice to convey the extent of your assistance and my gratitude, Thank you for everything.

Over the course of these three years, I responded to those inquiring about the Ph.D. experience, explaining:

"A doctoral thesis is the solitary confrontation with one's own limitations."

This Ph.D. manuscript is a testament to the collaborative effort of all those who have contributed to proving me wrong as I was truly never alone.

# Chapter 1

# Introduction

## 1.1 Background

### 1.1.1 Ubisoft

This PHD-Thesis takes place in a CIFRE agreement between the team **STORM** from Inria Bordeaux and **Ubisoft**.

Ubisoft is a video games editor created by Yves Guillemot and his brothers in Karantoer (Brittany) in 1986. Its first video game: Zombi, an action adventure game created by a team of 4 employees. (developed by Yannick Cadin and S. L. Coemelck, with graphics by Patrick Daher and music by Philippe Marchiset.) **Fig.1.1** was released in 1986 on AMSTRAD CPC


Figure 1.1: **Zombi starting screen, the first game edited by Ubisoft (1986)**

The company expanded over the years internationally, creating studio in Shanghai (1996), Montreal (1997), Barcelona (1998) and Milan (1998) and built a strong reputation as a video game editor over the years through the released of a lot of games (more than 700), impactful licences and, technological innovations from the 1990's till now. Notable games include : Rayman, Rainbow six, Splinter Cell, Prince of Persia, Assassin's Creed or Just Dance. **Fig.1.2**


Figure 1.2: **Examples of well known Ubisoft games released**

Nowadays, Ubisoft is considered as one of the biggest editor in the industry with **more than 21 000 employees** from 90 nationalities dispatched in 45 studios making it the **largest in house development team worldwide**. It has also diversified its operations, on various medium including films, series, comic books and even themed parks.


Figure 1.3: **Ubisoft studios presence around the world**

### 1.1.2 Video-Games Industry

Ubisoft is not the only company in video games that expanded massively in the past 30 years. Since the 1990's video games have gone from being children's entertainment to mainstream cultural media. **The global video game market size was estimated at 195.65 Billion $ in 2021, 220.79 billion $ in 2022** and is expected to reach 583.69 billion $ by 2030 [21] **Fig.1.4**, ranking it as the most important cultural industry in the World representing **more than Cinema and Music combined**.

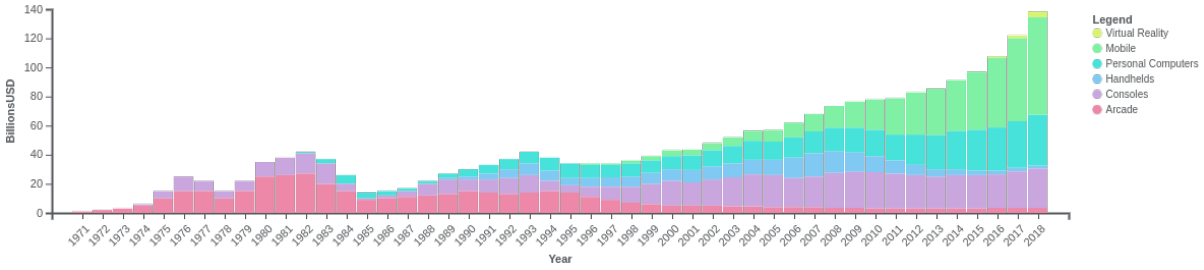



Figure 1.4: **Evolution of Global revenue estimate for video games between 1978 and 2018. The chart shows the impacts of the 1977 crash, the golden age of arcade games (1978–1983), the video game crash of 1983, the console revival (late 1980s), and the rise of mobile gaming since 2008. [46]**

By becoming mainstream in the past decades, the medium increased both the average age and amount of the standard player. The emergence of casual gaming and smartphone had a huge impact on its widespread with games like Candy Crush or Angry Birds. In 2022 it was assumed that roughly **3 billion people considered themselves as digital player[25] worldwideFig.1.5**

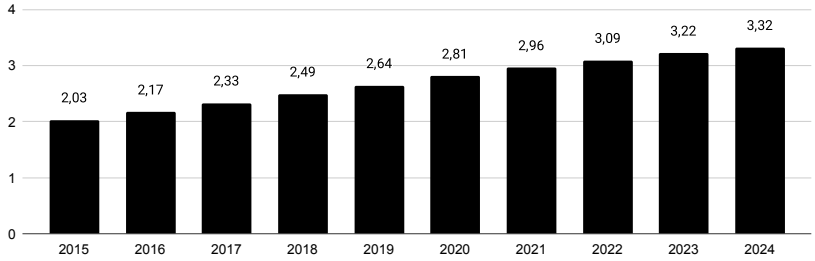



Figure 1.5: **Digital players worldwide evolution and projection in billions from 2015 to 2024 [25]**

With the rise of demand in digital entertainment, the video game industry diversified. As the industry evolved, the diffusion of video game also changed with platform as Steam or GoG allowing to buy dematerialized games, a broader diffusion and an easier marketing communication to the audience. The amount of game released yearly on Steam as shown on **Fig.1.6** greatly increased in the past 10 years. In 2022 it was estimated that **more than 5 million video games exist worldwide**, from high-speed racing adventures on mobile phone to perilous virtual dueling matches on high end Gaming PC.

Nowadays, video games are considered as one of the main cultural industry worldwide but it has been a long journey from the early consoles with the Atari 2600 to modern gaming devices such as PS5 or high end personal computers and Cloud Gaming.

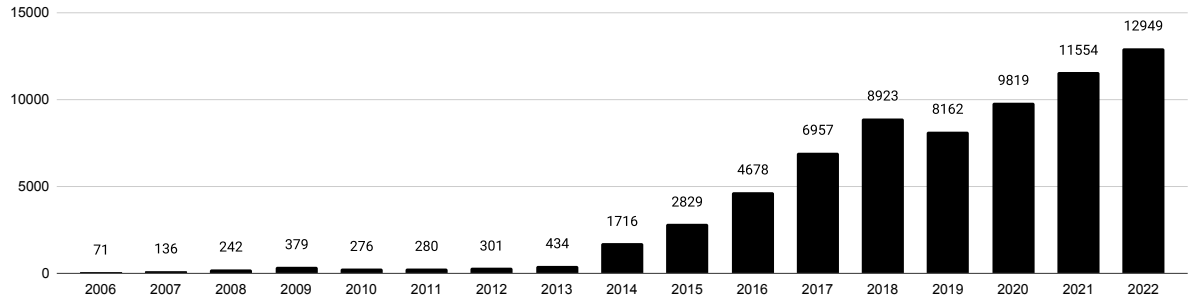



Figure 1.6: **Amount of Games Released on Steam per Year between 2006 and 2023 [38]**

### 1.1.3 Video-Games and Hardware Co-design

#### 1.1.3.1 Technical evolution

Figure 1.7: **First generation of console : The Atari 2600 (1977)**

The evolution of video games and hardware has been **a continuous intertwined process** and has greatly shaped the gaming industry into what it is today.
In the 1970's video game were mainly available on **dedicated devices** such as Arcade and early home consoles (MagnaVox Odyssey) with analog circuits. The limited technology available in term of computations at that time **strongly restricted the graphics and gameplay capabilities**. The use of floppy disks for distribution purpose were also restricting the amount of data available. However, some of the most renowned games in history were released at this period : Pong and Space Invaders, paving the way for the Golden Age of Arcade until 1983.

In the early 1980's emergence of more powerfull CPU allowed Atari to create **basic 3D graphics** using the **MS 6809** in it's Arcade terminal. Star Wars: The Empire Strikes Back was one of the first game to use this technology, allowing a better immersion and a new experience for gamers. At that time, CPU were handling graphism with development in assembly and C to provide maximum performances. Unfortunately, the **rapid growth of the industry** also led to a significant saturation, with too many mediocre games flooding the market. This resulted in the **video game crash of 1983**, marked by a significant decline in the popularity and profitability of videogames.

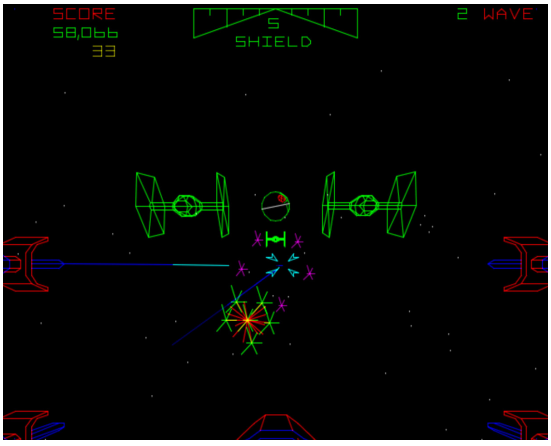



Figure 1.8: **Star Wars : The Empire Strikes Back on Atari (1985)**

The gaming industry **overcame this difficulty** with the introduction of more advanced hardware as new companies entered the market and new technologies emerged. One of the key factors that contributed to this resurgence was the **appearance of home computers**, such as the commodore 64 and the Apple II, making video game accessible to a wider audience. These home computers also allowed the development of more advanced video games, as the increased processing power and memory capacity of these machines provided complex gameplay and better graphics. Moreover the 1980s saw the rise of key players in the video game industry such as Nintendo releasing its iconic **Nintendo Entertaining System** (NES) in 1985. This was a major driving force for the revival of home consoles with a strong lineup of high quality games (Super Mario, The Legend of Zelda, Megaman etc.) and technological breakthrough with the **substitution of floppy disk for cartridges**, allowing better reliability, storage capacity, security and faster load time since the data were stored directly on the cartridge itself and could be immediately accessed by the hardware.
The NES also brought a key feature in video games with the introduction of **sound chips** in home consoles **(Ricoh 2A03)**. Initially, sound effects in video games were restricted to simple bleeps and bloops and basic musical capabilities. This chip diversified sound effect and music, allowing the creation of real orchestrations. This era saw the creation of musics widely considered as some of the most iconic soundtracks in video game history, such as **Koji Kondo's** work in "The Legend of Zelda" or **Hirokazu Tanaka** for "Tetris". This hardware evolution developed a whole new subject of matter in the video game development : the Sound Design.

In the 1990s the increased processing power of consoles with the introduction of 16-bit and 32-bit consoles such as the Sega Genesis provided video games with more complex graphics and gameplay including the possibility to **save player's progression** using **persistent memory systems** inducing the rise of RPG and history based video games. The **broadening of CD-ROMs** to consoles used as a storage medium for video games in the late 90s was a major turning point, by greatly **increasing the amount of data** available per game, which allowed more detailed graphics and gameplay.These new possibilities paved the way for a new type of game: the **AAA's** that are the games with the **highest development costs**. The first game to be considered as AAA was Final Fantasy VII released on PlayStation in 1997 with a development cost over 40 Million $. Moreover, **democratization of GPU's** in 1996 with the **Voodoo graphic cards**, allowing to offload the 3D rendering tasks to a dedicated hardware accelerator, greatly improved performance and allowed for more realistic and complex 3D graphics to take place. In 1997 3Dfx released **Glide** [3], a subset of OpenGL designed to take full advantage of the hardware capabilities of the Voodoo Graphics cards, providing a high-performance, **low-level interface for game developers** to access the card's 3D graphics acceleration capabilities thus **avoiding cumbersome C or assembly programming**. These two technologies also **reduced the stress on the CPU**, enabling for more computations to take place, emergence of better NPC, more complex game-play.

It is also important to notice that even if Glide was available to ease the graphic offload, this API was specific to Voodoo graphic cards. Every constructors had its own API and game were implemented in several version (One per GPU constructors) in the same manner that games were specified for consoles. In this context, some video games were only available for certain brand of GPU. In contrast, **Direct3D** was released by Microsoft in 1996 and was a **more general-purpose graphics API**. While it did offer some 3D acceleration features, it was not optimized for specific hardware. Additionally, construc-


Figure 1.9: **Comparison of Quake classic implementation (left) and with OpenGL (right)**

tors developing their own API, were able to tightly integrate it with their hardware and optimize it for maximum performance. Which was a popular design decision among game developers who wanted to push the boundaries of 3D graphics in their games.

This decades also marked the **emergence of multiplayer** with the introduction of local split-screen, Local Architecture Network (LAN) and even the first glimpse of Online Multiplayer on home computers leading to the online revolution of the 2000s.


Figure 1.10: **Screenshot from World of Warcraft 2004**

The **emergence of internet** drastically changed video games. First of all the possibility for developer to **gather a great amount of players** at the same time created a new genre, **the Massively Multiplayer Online Games** (MMO) with it's most iconic game World of Warcraft (**Fig1.10**), contributing on both the social and competitive aspects of the medium. Then the early stages of **digital distributions** provided better communication and access to video games for the players. Finally and maybe the most important side of this technologic advance, internet provided the developers the capacity to **update the games and add content even after the release.**

In 2001, the **Power4** released by IBM as the **first non-embeded microprocessor**

**with two cores** represents a shift. The rise of multi-core CPU changed the way video games handle graphism again. At that time, the main issue was that while games were already using GPU to render all the 3D, they were not fit to run with several threads on the CPU. In order to use this second thread, game developer **separated the logic from the graphic frame and bufferized one frame**. By doing so, a thread could run the game as usual by computing all the logic and the second thread would be in charge of handling data transfer to the GPU to construct the image to be drawn to the screen.
The 7th generation of consoles released between 2005 and 2006 saw significant differences on the hardware architecture between the 3 major constructors : Sony, Microsoft and Nintendo.
**Sony's PlayStation 3** (PS3) was built around a custom Cell processor, which featured one general-purpose PowerPC core and eight co-processors known as **Synergistic Processing Unit** (SPUs). The SPUs were designed to handle specific tasks, such as audio processing or physics calculations, and were optimized high speed vector processing. This architecture enabled very high performance but it's complexity made it difficult for 3rd party developers to specialize computations and use . This resulted in under performing video-games for most of multi-platform release. On the other hand **Microsoft** opted for an architecture **closer to Home Computer** with a triple core CPU and a GPU providing less performances but more accessible for developers already used to implement video-games for PC architectures. Finally **Nintendo** chose a classic architecture for the Wii but decided to **focus on motion gaming** providing more accessibility to video games since the simple and intuitive gestures required could be more accessible than complex button combinations.

In the Early 2010 **SSD** (Solid State Devices) became more affordable and their capacities started to increase. Before that, SSDs were mainly used in high-end systems or in specialized applications where speed was critical, due to their high cost per gigabyte compared to traditional hard disk drives. More affordable SSDs enabled both PC and consoles to adopt them. this greatly **impacted loading times** since data can be accessed much faster. In addition SSDs also had a significant impact on game asset streaming, allowing games to stream in **high-quality textures, models, and other assets** more quickly, resulting in a better overall gaming experience and the possibility to **extend map size** way easier. We noticed during this decade an homogenization of consoles architecture with Sony renouncing to the SPUs. Knowledge and experience in developing on such systems also allowed **software development Kit** (SDK) and **Tools** used to develop video games to drastically improve. This resulted in significative increase in performance directly **impacting the aspect of video games during consoles exploitation lifespan** as seen on **Fig.1.11**


Figure 1.11: **Comparison of Fifa 2015 (left) and Fifa 2021 (right) on PS4**

Moreover the computation power of GPU and CPU greatly increased sustaining the computation growth to take place on both these components. This allowed for real **technological shift** from a **console generation to another**[41] marking a gap of 6-7 years between the phase. One of the interesting fact was also that **during transition phases between generations**, video game developers had to adapt in order to **allow the game to run on older and recent platforms.**

Figure 1.12: **Comparison of Assassin's Creed Black Flag on PS3(left) and PS4(right) (2013)**

All these technological evolution induced as it became more complex the introduction of new matter of concern in video games development, organizational requirements to produce video games evolved continuously to fit the need of production.

#### 1.1.3.2 Organizational evolution

In the 1970s, most of games were **designed by single individuals or small teams**, with few developers and a designer devoted to pixel art and animations. This organization remained until the end of the 1980s but technological evolutions in the video game industry had a profound impact on the organization and structure of video game companies. Thus leading to the **creation of new roles and responsibilities** as video games became more complex and vast.

> *It's impossible for one person to have the multiple talents necessary to create a good game*
>
> Yakal, Kathy,*The Evolution of Commodore Graphics, 1986*[45]

This led to the specialization of roles within video game development teams, with employees focusing on specific aspects of the game.

The primary area of specialization in video games was of course **artistic jobs**, allowing a greater focus on creating visually immersive game environments. The demand on high

quality graphics induced even further separation of concern with the creation of specific jobs on a wide range of subjects including conception roles with the **Concept artist** responsible to conceptualize and set the tone and style through the creation of sketches and illustrations and **UI/UX artists** in charge of all interactions between the game and the player. **Fig.1.13** provides an example of concept art and 3D modeling from Ubisoft.

Figure 1.13: **Illustration of concept art (top right) and 3D Modeling for the game Beyond Good And Evil presented in 2017**

There are also artist in direct contact to in-game artifacts for instance **3D Artists** in charge of all three dimensional assets and environments, **VFX artists** managing lights, explosions, particle effects and other visual effects and **Animators** responsible for all in game animations and cinematics. One of the earliest examples of this specialization was also the role of **game designer**, responsible for the design and direction of the video game, creating the game's rules, levels and objectives. By doing so, the game designer has a direct impact on balancing challenge and overall engagement and entertainment for the player. In order to make the game more immersive, a lot of work is put in **Sound design** with specialists dedicated to create **sound effects or compose musics**. This aspect of the game represents such an impact that it is now rewarded in international events as the Grammy Awards. Considering the size and complexity of video games, another need appeared, the **quality assurance** (QA). This process is ongoing throughout the entire development cycle of a game, from conception to the final release. It aims to ensure the quality of the game by **testing and identifying bugs or performance issues**. QA testers will play through the game repeatedly, trying to trigger any potential issues and documenting all the steps to reproduce problems. This area is essential as even minor errors or bugs may significantly impact player's experience, even making the game unplayable. Given the increase number of persons working jointly to develop video games, it was necessary to also change the habits of **management and organization of the project**, by introducing dedicated management roles to lead the development teams. The **project manager** is supposed to handle all the different aspects of the development, ensuring all the different team are working toward the same goal, monitoring the budget and that the different milestone are respected. This aspect is very important since all teams are working independently, some lock may occur blocking the whole development team. In order to create complex systems conceptualized by game designers. The role of **gameplay programmer** emerged. Its role is to implement combat mechanics, character controls and other features such as artificial intelligence behavior. In the same manner, in order to keep away game developer from low level development, video game company developed a **core software** called the **game engine**. The game engine is a complex tile of video game creation and, requires multidisciplinary teams to operate, among which **3D developers, architects, optimization specialist and experts** on every area it impacts.

## 1.2 Game engines

In the early 1990s, video games production was "industrialized" as it became a mainstream cultural medium. The urge to create video games faster led to specific software development.

### 1.2.1 What is a game engine?

Primary, to ease the development process and since C++ was not already in use to implement video games (thus the absence of objects), **common architecture of programming were developed** allowing to map objects such as enemies, textures or weapons using macros. This technology was perfected, providing developers the capacity to **create several maps for a game** much faster than before. This technological evolution also enabled for developers to **create (very) similar games** from scratch by updating the different sprites representing enemies, texture and redesigning maps as seen on **Fig.1.14** with the Doom Engine. For over 20 years, **these software were continuously improved**, providing nowadays a high level of interaction through **visual tools and scripting process** allowing for non-developers to take direct part in the development process and create from scratch any type of games. There are legions of **different game engines** with their own specificities providing benefits depending on the type of game being developed. However the **underlying technology and operation remains the same**.


Figure 1.14: **Example of two different game created on the same early engine Wolfenstein 3D (1992) left and Doom (1993) right**

The **Game Engine** is a **Core Middleware** development platform, its roles are multiple during the different phases of the life of a video game. It's primary function is what we call the **Editor or Tool Mode**. It constitutes the foundations of the game to be created and consists in **proposing a framework and a technical pipeline** for the whole development team **to create and test video-games**. It Provides simplified access to the development process and a higher-level abstraction that allows developers and artists to focus on creating game content rather than dealing with low-level technical details. It

integrates consistent, reliable, and well-documented set of visual tools and scripting methods for both developers and non developers to interact directly with the game in creation. It provides a clear **separation of concern** between Game developers, Artists or Sound Designers **creating the game** and Engine and 3D Developers enhancing **the underlying technology** in order to deliver optimal performances and providing the game with as much computations resources as possible.

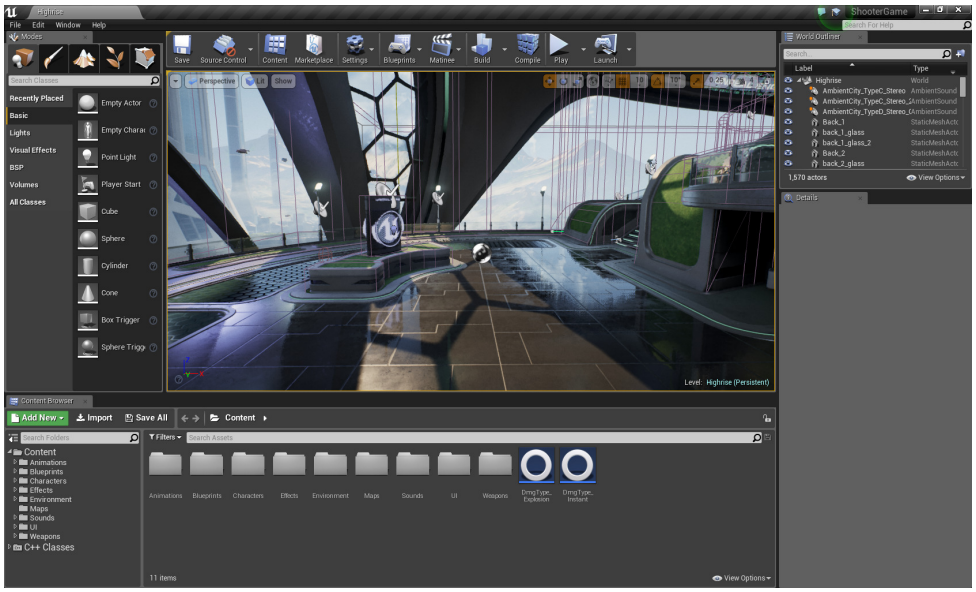


Figure 1.15: **View of Unreal Game Engine in Editor Mode to create scenes and interact with the world in construction[40]**

Considering the complexity of modern games and in order **to minimize development costs** for the company, game engines are often developed in an **ongoing process over a decade** through frequent updates modernizing its feature and its architecture. It can also be specialized to better fit the needs of the company. These MiddleWare can be licensed, such as Unity, Unreal or Open 3D Engine, or kept within the company as important assets. This approach of **incremental development** provides a lot of advantages since it enables to organize and **speed up development process** by re-using the same game engine to create multiple games and **avoid the re-implementation** of several aspects of the video game. It also significantly **reduces the time devoted on testing and debugging** since most of the parts of the core systems have already been tested in past iterations.

The second mode of the engine occurs when the game is ready for testing and deployment and is called the **Game Mode**. This represents the state of the **engine that runs the game** itself. It is optimized for performance and memory usage, and typically includes only the core components, features and assets that are required by the game to run fully without the possibility to edit or access private data and mechanisms of the game itself. This is the part of the engine that is **shipped to the player** with the game and the one we will **focus on in this manuscript as it requires optimal performance** in order to provide the best experience for the players.

Before focusing into understanding what are the different components of the game engine and how it runs a video game, we must understand **how it interacts with its environment**. When a game is running on a game engine, its role is to **intercept the**

**different inputs** generated by the players through peripheral devices. These inputs can be vocal interactions when the player is speaking or singing (depending on the type of game) or physical interactions using Joystick, Mouse and Keyboard or even Camera. The game engine then **translate these inputs into actions** interpreted by the game using the different resources available depending on the device the game is currently running on. The engine will then, with the computation power of the device, **determine the new state of the game** and, **construct a set of outputs**, in terms of image and sounds. These outputs are then **delivered to peripheral devices** such as screens, virtual reality headset or speakers as shown on **Fig.1.16**

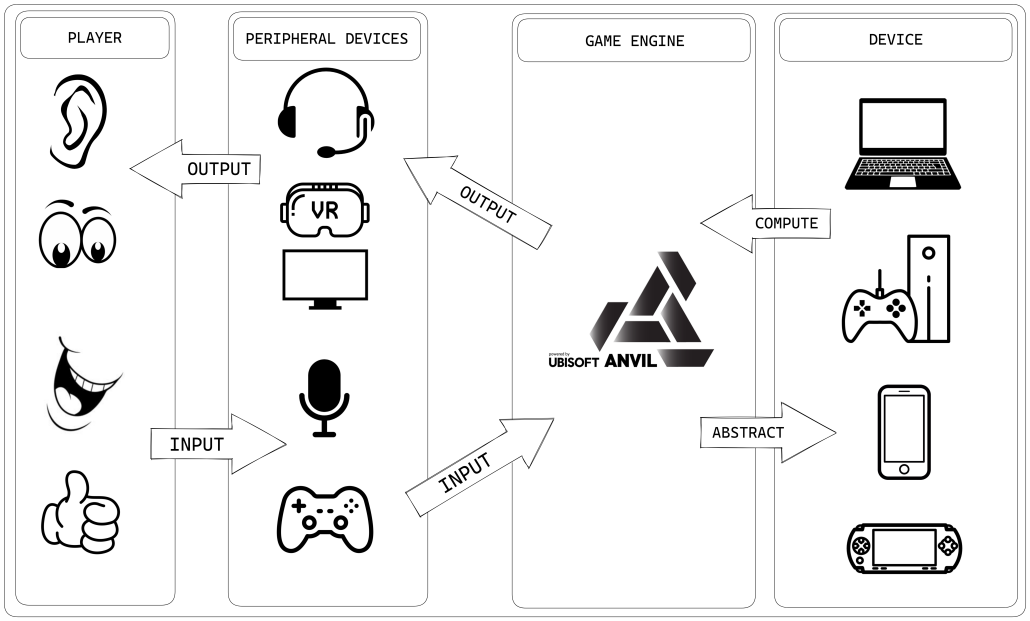



Figure 1.16: **Description of the Interaction between Player, Peripheral devices, Game Engine and Devices**

Since most video games are supposed to be **interactive and real time**, in the meaning that it should continuously deliver the images to the screen. In order to induce a feeling of motion for the player and due to our own perception of continuity, the cycle described earlier must be fulfilled at a certain rate. Just as cinema that usually display 24 images per seconds when a film is projected, a video game has to keep up a certain rate of image making. These images is what we call **Frames**. If the game engine **fails to create images** at the expected rate, the game will skip one or several frames before rendering the next one, **resulting in a stuttering or bumpy animation** that can be very noticeable. This has a significant impact on the player experience, making the game less responsive and harder for players to react to in-game events. This may lead to frustration and potentially impacting their ability to progress in game. This phenomenon is known as a **Frame Drop**.

The amount of Frame per seconds(FPS) required for a video game to be considered as fluid has greatly evolved. The **current standard** is set at a rate of **30FPS** for devices with "low" computation resources, and **60FPS** for better configurations. In competitive area of video games, the framerate constitutes a crucial element of the gameplay since it directly impacts the player reaction time. In this context most of pro-players are running the game at 120FPS, which represents a frame every 8.333ms and up to 240FPS representing a frame every 4.166ms.

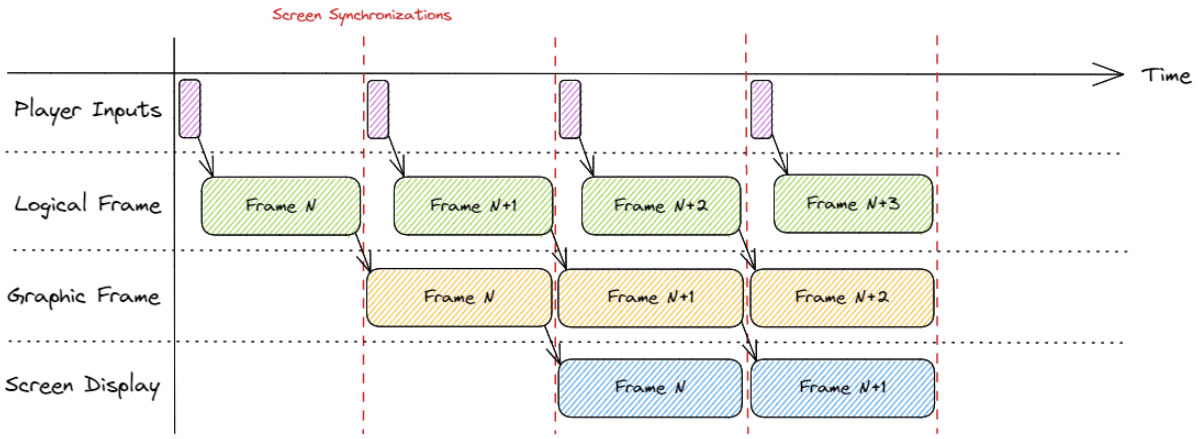



Figure 1.17: **Description of the Game Engine Pipeline from Players Input to Screen Display**

In order to **construct the image** to be exposed to the player, each frame undergoes **3 levels of pipeline** on the different treatments a frame must undergo. These treatments being originally sequential, we use a system of double buffering to store different versions of the image, this allows to handle all these treatments in parallel since they occur on different frames. **Fig.1.17** represents the different phases of the pipeline.

The first treatment consists in **input handling and logical frame computation**. The logical frame is responsible for **processing the game's mechanics**, including physics, and AI. This processing requires the use of the CPU, which performs calculations and updates the game state. This resulting game state is saved and **forwarded for the graphic frame**. It is responsible for the **rendering of the image** including geometry, textures, and lighting effects. This processing requires the use of the GPU, which performs calculations to generate the visual output as well as part of the CPU to transfer the required data to the GPU. The construction of this image is done in the rendering buffer, a different buffer from the screen (also called display) buffer that is used to store the pixel data to be **displayed on the screen**. This technique helps to prevent flickering[1] and tearing as shown of **Fig.1.18** that can occur when the display buffer is updated while it is still being rendered. By rendering to a separate buffer and then swapping the buffers once rendering is complete before the screen synchronization, the final image is displayed seamlessly without any visual artifacts.

To gain a deeper understanding of game engines, it's essential to explore their structure and components in detail. While we have already discussed the high-level concept of these middlewares, a game engine is actually a complex system that consists of multiple modules working together in tandem to create an immersive and enjoyable gaming experience. Breaking down theses components and their roles can provide a more in-depth perspective on their functionality.

### 1.2.2 Structure and Components of the Engine

To better understand the various components of a game engine, we will examine three distinct screenshots issued from Assassin's Creed Valhalla, each taking place in both different phases and contexts of the video game. By exploring these examples, we can gain insights into how the various components of a game engine work together to bring a video game to life and see how a game engine is not just a single piece of software but

Figure 1.18: **Example of Tearing in Portal (2007)**

rather a complex system that incorporates various components, each with its own unique function and purpose.

The Screenshot Fig.1.19 (a) represents an action phase of the game with NPC fighting around and an apparent quest to fulfill. (b) is a contemplation phase with very far sight and a lot of detail to draw on screen and finally (c) is a Menu phase with a detail 3D rendering of a character with inventory and resources management in game.

The first role of the engine we will address is easily noticeable on the 3 screenshots and is linked directly with graphism.

**The graphic engine** is responsible for **rendering the game world and its objects** on the screen. It is designed to handle high-quality graphics and visual effects that are expected in a AAA Game. It is composed of different tiles. The first one is the 3D Modeling that creates the game world itself such as characters, buildings and terrain. All these elements together **produce a 3D scene** in which the camera can be positioned depending on the player position and of course it's interactions with the game. **Fig.1.20** illustrates the state of the game after this phase.

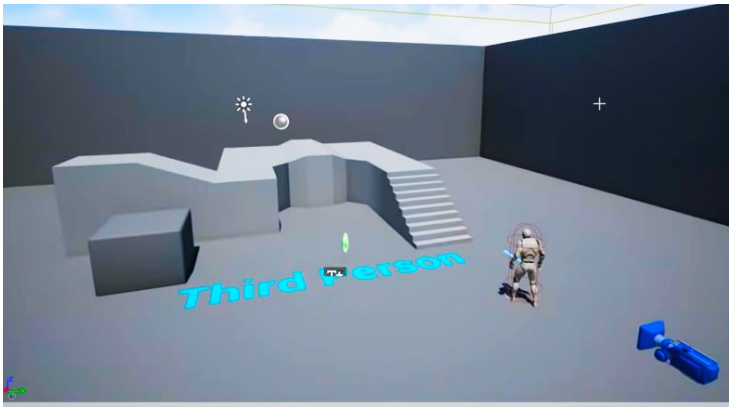
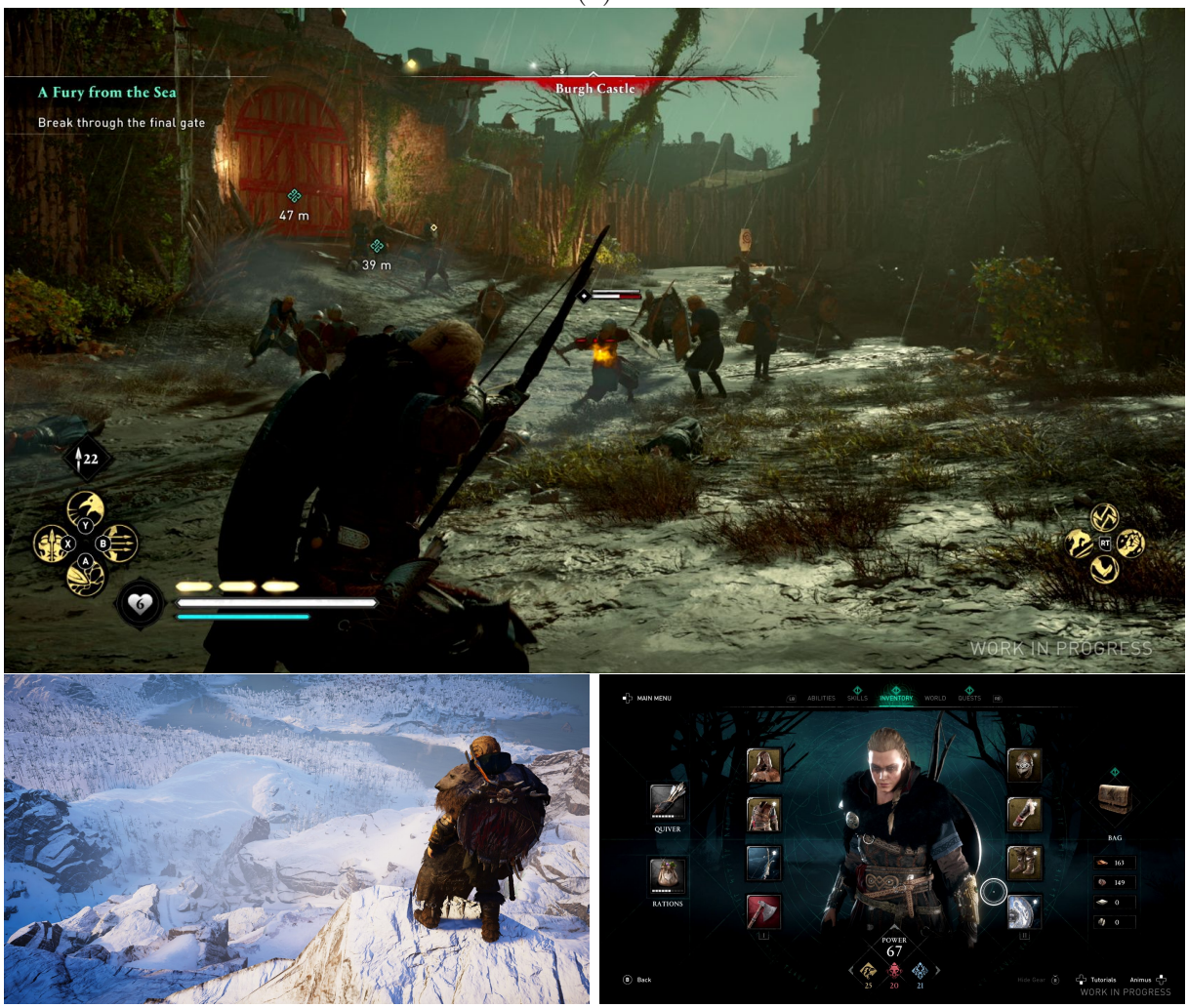



Figure 1.20: **Illustration of very simple world and player with the position of camera in Unreal Engine 5 without textures**

Based on all these different aspects, the rendering occurs in order to **apply all corresponding textures** to the image, and **particle effects** such as snowflakes **Fig.1.19 (b)**, sparks, smoke or fire to create visual effects in the game. The **lighting and shading** of the scene that simulate the way that light interacts with all the objects in the world including reflections, global illuminations and, resulting shadows occurring at that moment.

(a)




(b) (c)

Figure 1.19: **3 different screenshot of the Video Game Assassin's Creed Valhalla during different phases, Action (a), Far Sight and Observation (b) and Menu and Inventory (c)**

**Fig.1.19 (a)** is a great example of how light and shadows are projected depending on the 3D models. Finally, the engine needs to create a projection of this 3D World and all the different visual effects that occur from the Camera point of view in order to **generate the final 2D image** that is displayed on the screen.

Once this image have been created, it needs to be completed with another key visual component of the engine, the **User interface (UI)**. The User Interface aims to **provide feedback information to the player**. We notice in **Fig.1.19 (a)** that display on the screen, the different actions available, the amount of arrows, life, stamina but also the different objectives to fulfill the ongoing quest. On another hand and still regarding the UI the screenshot **(c)** shows the Menuing in Assassin's creed Valhallah with the visual of the different items equipped by the player, but also various important options such as the visualisation of the Map, Game Option etc.

A second very important aspect of a video game is the **Sound**. The sound engine handles all audio treatment of the game, and support advanced features.

There are different kind of sound used in a video games. Music that is a **continuous sound** runs during the game in order to set a mood, sound effects that are **short audio clips** used to enhance in game experience such as footsteps, explosions and are most of the time triggered by in game actions. And finally **Voiceovers** that are audio recordings of character dialogue or narration that are used to provide informations to the player or construct the plot of the game. These voiceovers are most of the time triggered through gameplay scripts.
The sound engine must play back all theses differents sounds depending on the various triggers. It also **applies transformation** depending on the Audio scripts by combining multiple audio sources and applying effects such as **volume, panning and reverb** to create the final audio output. It can also, through **audio spatialization** simulate the way sound behaves in a 3D space, including the way it is affected by the position and orientation of the listener and the sound source.

In order to **construct a coherent world** for the video game, the engine must respect and **handle the different physical rules** that apply to the world. This is the role of **the physics engine**. It provides computations regarding the **physical interactions of characters** (players or non players), objects or simply **forces** that apply through different tiles, among which the **collision detection and response** that aims to detect and handle the consequences of a collision between two or more objects in the game world such as bouncing or deflecting off of each other.
These collisions will have various consequences depending on the nature of the object. We denote three main characteristics for such entities, the first and easiest one to compute are the **Rigid bodies**. In order to simulate the behavior of such entities, we only take into account its shape, mass, velocity and acceleration. The second one is way more complex and are the **Soft Bodies**. These are the entities that can change shape with the different interactions, we have to take into account the resistance of the material, its weight but also all the different deformations that can occur and that should remain in time, to give an example of well known soft body object we can cite cloth or destructible environments. Finally the physics engine also need to handle **Fluid Dynamics** in order to simulate fluids or gases. This includes the way that they flow, collide and interact with other objects. **Fig.1.21** illustrates the simulation of water in a recent video game.

Figure 1.21: **Water behavior simulation in Sea Of Thieves on PC with RayTracing (2022)**

Finally the physics engine handles all the different forces that apply to the different objects and characters in the game and beyond collisions, the engine also handles the simulation of external forces that can affect the movement and behavior of objects in the game such as wind, gravity or magnetism. But also the **constraint to simulate physical connection between objects** such as ropes or even the contact force of a character with the ground. It is primordial to notice that **even though the physics engine is supposed to compute physical events, it does not need to be precise nor exact**. Different tricks are used to give the **impression of genuine interactions** without the amount of computations required as this would necessitate way more powerful devices.
In order to create movements and behaviors of entities in the game world, the engine also need to handle animations of both objects and characters as seen in **Fig.1.22**(thus skeleton) through scripts and physical reactions such as Ragdoll

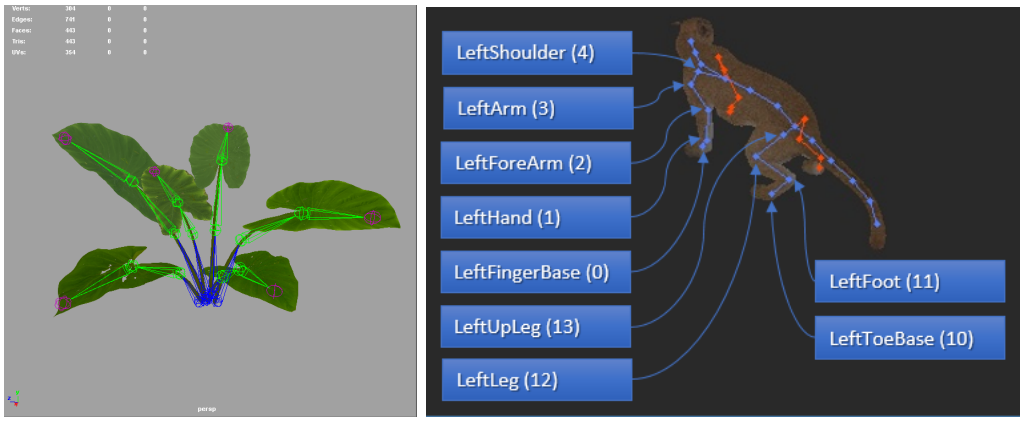



Figure 1.22: **Example of different skeletons created to animate and compute forces applied to objects and characters used by Ubisoft**

We can also cite in-game cinematics and **high-definition pre-rendered video replay** to complete the panel of the different actions engine must handle. At that point we described all the required mechanism to be able to construct an animated image with audio management. Nevertheless it is necessary to notice that engine also deal with **low level features** to manage hardware that will run the game. In order to facilitate understanding we will not go in too much in detail concerning these area but we need to explain few mechanisms that'll be crucial for the rest of the manuscript.

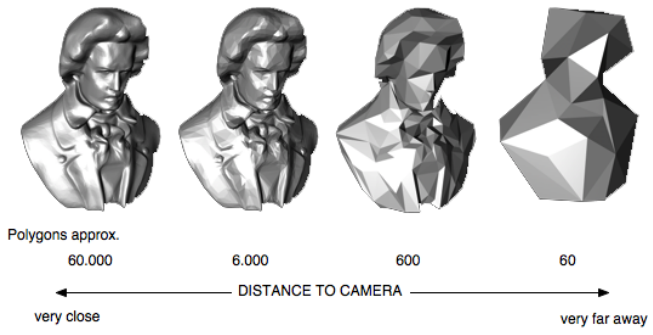



Figure 1.23: **Illustration of Level of Detail on a 3D Model representing a Bust [14]**

The first area we need to address is **Memory Management and Asset Loading**. In order to be able to create the world designed by Artists and Game designers, the engine needs to be able to handle organization and loading of game assets, such as models, textures, and audio files. The large numbers of assets and their size require to support efficient resource management such as FastLoad, Asset Streaming etc. This represents one of the main technical issue of game engine. **Loading data takes a lot of**

**time** and as seen in **Fig.1.19 (b)** if the field of view is relatively high, the amount of detail to load is unreasonable. One key mechanism we use in this context is the **Level of detail (LOD)** as shown in **Fig1.23**. This method consist in **adapting the representation's complexity of a 3D model depending on it's distance to the player**. We reduce the amount of polygons that defines a 3D object or terrain. In this context the reduced visual quality of the model is often unnoticed because of the small effect on object appearance when distant or moving fast. Most of the time, the LOD's or predefined model used depends on the context, but there are also other techniques such as Tesselation (illustrated in **Fig.1.24**) that allows to change dynamically the amount of vertex for a given entity.

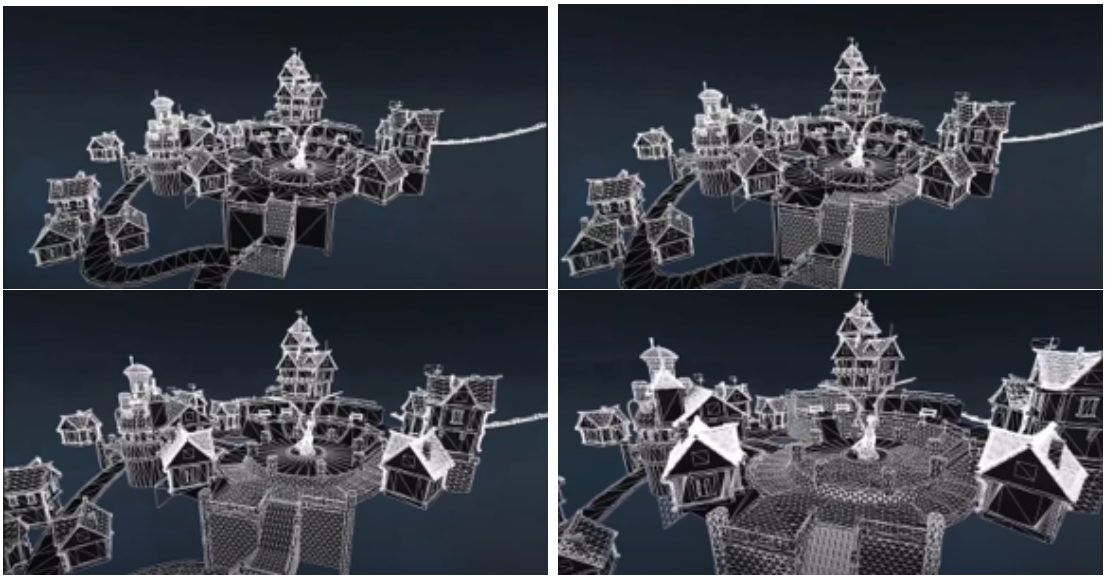


Figure 1.24: **Example of Tesselation of a same world with different viewpoint**

Although most of the time LOD and Tesselation are applied to geometry details only, they have been applied recently to shader management and texture maps.

We covered most of the engine's components yet, a fundamental aspect that makes up a video game still remains to be discussed. **The gameplay** handles rules that define **how the game is played**, the challenges and, objectives that the player must complete.

The gameplay is a crucial component of a game engine, responsible for organizing and utilizing various engine aspects. It **interacts with all engine parts** and performs three primary functions: **constructing the world, defining player interactions with the created world, and managing the game reactions to player actions.**

In order to construct the world the gameplay tile handles the different assets and data that make up the game such as levels, enemies, weapons, and other items created by the artists. It also orchestrates how these components behave and interact together to generate a coherent world and an immersive experience for the player.

Defining the different interactions available for the player with assets composing the game involves creating key mechanisms using the physics engine such as character, movements and, combat.

Finally handling and creating the reaction of the game to the player action requires creating scripting methods to compute environmental response, event triggering and dynamic changes in the game. Moreover, it simulates the behavior of Non Player Characters (NPC) and other "AI-controlled" elements.

**NPC behavior** is a complex matter and can be simulated using various strategy. For instance with a **set of rules or conditions** that determine how the AI should behave in different situations. For example, a NPC may be programmed to always attack the player if they get too close, or to run away if their health drops below a certain threshold. All these different rules are **organized in a Behaviour Tree** defining the different behaviors of an NPC as a hierarchy of interconnected nodes. The NPC will then follow the path through the tree that corresponds to its current state and the conditions of the game world.

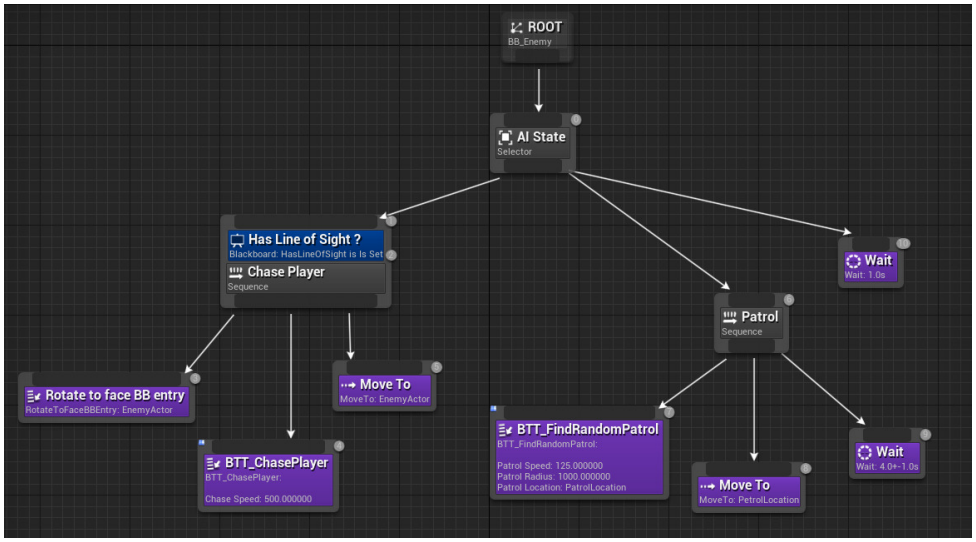



Figure 1.25: **Example of Behavior Tree creation in Unreal Engine 5**

Another method that emerged lately are neural networks allowing AI to learn and adapt over time to player's action. This enables the AI to make more complex and nuanced decisions based on past observations. Once a decision have been taken for the NPC, the game engine must calculate the most efficient path to follow in order to reach its destination. This process of pathfinding involves analyzing the layout of the game world, identifying obstacles and potential hazards, and determining the optimal route for the NPC to take. The game engine uses various algorithms and data structures to perform this task, such as A* algorithm and a grid or navigation mesh. This enables the NPC to move smoothly and intelligently throughout the game world.

**Networking will not be addressed** in this manuscript, as we focused on offline solo AAA's games. Network management is one of the area of the engine and affects several tiles as video games states needs to be synchronized over several machines.

We tried in this section to give hints on the different computations required to create interactive 3D simulation. In reality, a game engine is much more complex and is composed of way more different tiles. **Fig.1.26** describes the real architecture of a game engine with all the different tiles. Theses **layers are co-dependent and their executions are structured by phases organized around physics treatment.(PrePhysics, DuringPhysics, PostPhysics)**

The game engine is a **multi level architecture** with the higher level based on the lower ones. The lowest layers manage resources, control code independence to the platform and orchestrate parallelism on the device. In order to **optimize parallelism** and provide better performance, the engine is organized as a **task-based system**. The **different tiles composing layers** above "Core Systems" **generate tasks or small task**

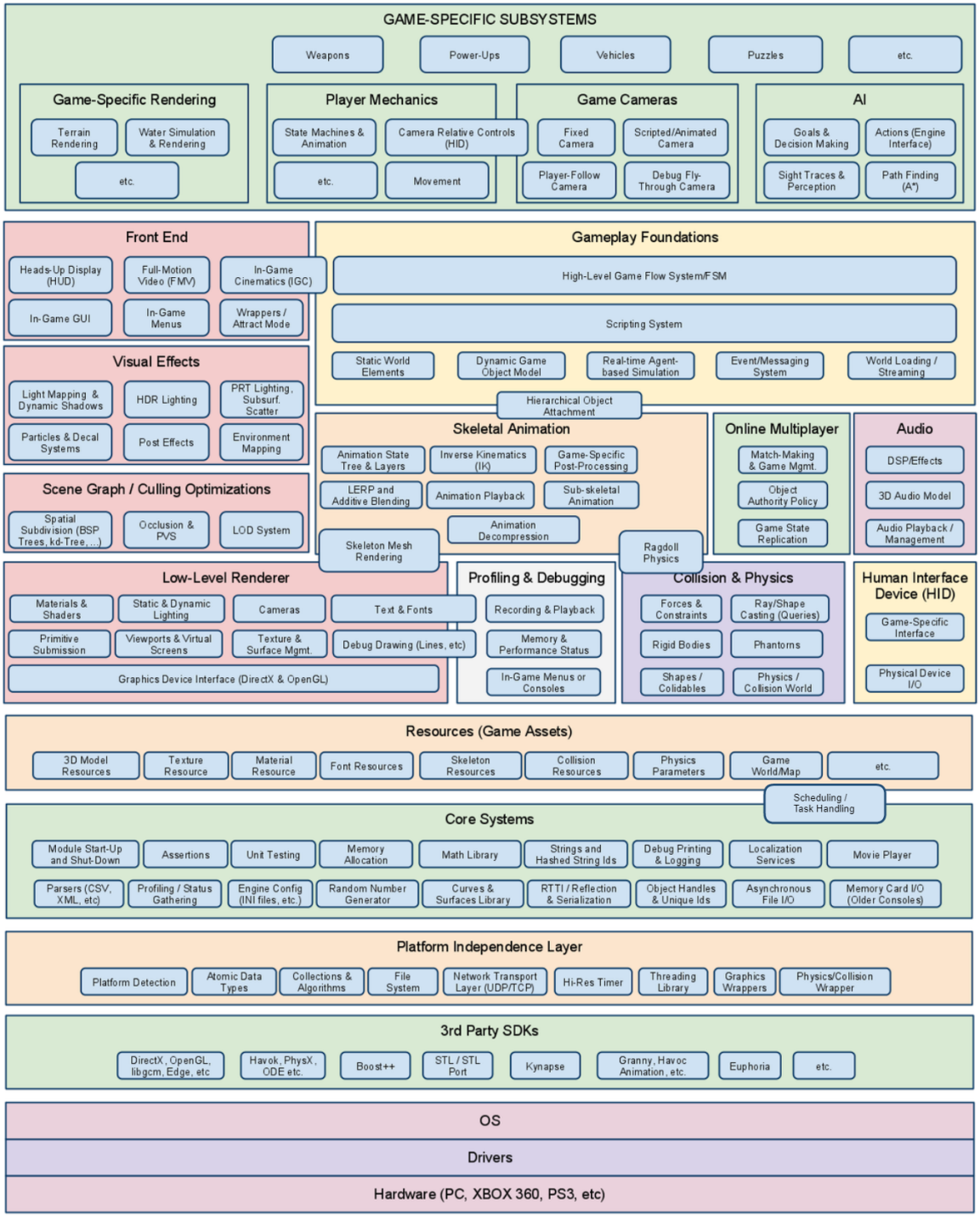



Figure 1.26: **Detailed Architecture of a Game Engine [22]**

**graphs**. These structure are then organized in a Directed Acyclic Graph describing data dependencies.

**Game engine is also in charge of the scheduling.** This refers to dispatching the various tasks composing video game task graph on available resources of the running platform. It aims to be as efficient as possible, taking into account architecture of the device as well as parallelism available on the game.

It is important to specify that **this task based system is not implemented in every engines** but in recent years, this code paradigm became more popular.

## 1.3 Scheduling Problematic for video games

When using a task programming paradigm, **identifying and prioritizing the tasks** to be distributed among available resources is primordial to provide maximum performances to the execution. However, in order to explain how scheduling in video games is handled, we first need to describe few specifities of the problem.

### 1.3.1 Hard Real-Time and Soft Real-Time system

In its early life, video game was considered as a **Real time system**, meaning computation were to be computed within a time constraint, by **failing to respect this limit, the integrity of the system is compromised**. An example of real time systems is the Pace Maker, by failing to fulfill required computation, it may result in potential threats for human lives. Even though the result was less catastrophic in early video games, the process of rendering images on the screen was such that missing a deadline could cause the game to crash, or result in a partially rendered or entirely black screen.

Nowadays, with the double buffering technique referenced in **Section.1.2** and with the evolutions of resources, video game are not considered anymore as Real Time Systems but **Soft Real Time Systems** meaning computation are to be computed within a time constraint, but **failing to fulfill the requirement does not compromise the integrity of the system**. Video games are considered Soft Real-Time Systems because they have specific timing requirements as Frames are supposed to be computed in 16.6ms or 33ms depending on the platform, but occasional delays or missed deadlines do not necessarily result in system failure but will cause a Frame Drop as explained in **Section 1.2**. Moreover, while it is essential to maintain a high level of responsiveness and performance, some tasks composing the frame could be defined as **Skippable**. These tasks, are particular as they can be **deferred or delayed** without significantly affecting the game's overall performance or the player's experience. Examples include loading assets, pathfinding, or certain animation computations such as explosions. However, it is important to manage these tasks carefully to ensure they are eventually executed and do not accumulate, leading to more severe performance issues.

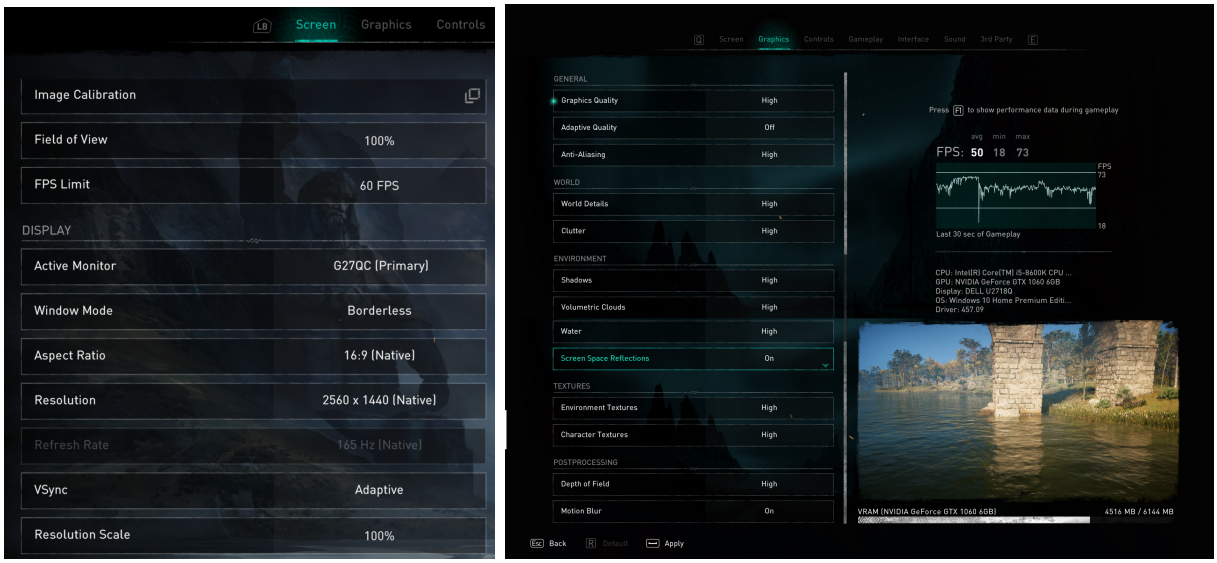
### 1.3.2 Current Scheduling in video games

In order to optimize game performance and guarantee a certain frame rate for every device (30 Frames per seconds for oldest Generation, 60 Frames per seconds for newest one) developers need to carefully manage the allocation of computing resources. One of the first issue game developers are facing is to optimize video games for a **wide range of**

**hardware configurations** from devices such as the Nintendo Switch embedding a CPU ARM 4 Cortex-A57 cores and a GPU Nvidia GM20B Maxwell-based capable of reaching 500 GFLOPS to the PS5 with a CPU Custom 8-core AMD Zen 2 and a GPU Custom AMD RDNA 2 with 10.3 TFLOPS peak or custom PC. In order to adjust the quality of the game and the amount of computation per frame, video games usually include a possibility to configure a great range of options.

These options are usually pre-configured depending on the device the game is run on but can be changed in order to handle the quality of different area of the graphism such as the depth of field, or reflection and shadow quality of detail as seen on **Fig. 1.27 (b)**. These options allowing to change the quality of shadders, computation methods or simply the resolution of the image constructed have direct impact of the Frame rate depending on the amount of computation resources available.

Usually it is also possible to configure the amount of desired FPS **Fig.1.27 (a)**. This can be useful to cap the amount of images and guarantee no frame drop will occur with the current graphic configuration. It can also be increased (without any limit) to have a smoother execution but may require to reduce the quality of the game graphics. All these **settable options mostly refers to GPU computations**, the only option that as impact on the CPU are the amount of FPS and sparse computations that can be offloaded from GPU to CPU. The main reason for this is that CPU action mainly occurs regarding the logic of the game and techniques allowing to create this sort of quality level concerning logical tasks have not been implemented yet.




(a) (b)

Figure 1.27: **Static tuning menu for Assassin's creed Valahallah**

In order to manage the amount of computations on GPU, **a budget is fixed per area of the graphic loop** and is decided by the project manager or game designer (a human). This budget is determined per game and is **a complex arbitration**. The different concerns to construct game's world and display drawn image must compute its tasks respecting the pre-determined budget. This sets limits regarding the amount of objects, lighting effects or particles the game can draw for a given scene, thus while constructing the game, developers and FX Artists have to take these into account in order to manage the computation's load per frames.

CPU computations are handled in a different way. CPU tasks can be classified in two different small graphs, the **game engine loop** responsible for a wide range of tasks including game logic, physics simulations, AI, input handling, and audio processing, among others and the **graphic engine loop** in charge of handling and transferring data to the GPU for the rendering of the frame. As shown in **Fig.1.17 these two different loops are independent** as they focus on two different buffers but need to be synchronized at the end of the frame to swap the buffers safely.
As a matter of facts, **engine loop tasks cannot be "tuned" in the same way we do for GPU graphics tasks** since setting a fixed "Quality" for such tasks, for instance reducing the complexity of all AI calculations, level of physics simulations or audio processing impacts how the game's world react to players interactions. By changing drastically these interactions, this **disturbs gameplay mechanics** thus the game itself. Moreover, these changes could have catastrophic consequences regarding game scripting and generate bugs.
Nonetheless, it is important to specify that through scripted and pre-determined use of such quality scaling, it is possible to free up CPU resources in order to maintain a smooth gaming experience. For instance we can cite in Assassin's creed Unity that simulate a game world set during the French Revolution, with dense crowds filling the streets of Paris. In order to do so, it respected a set of pre-determined rules to optimize the CPU usage. By reducing AI Quality and animations for far ranged NPC that were not to interact with the player immediately as seen on **Fig.1.28**[16].

Figure 1.28: **Example of crowd simulation using reduced AI quality and animations**

These kinds of tricks are only possible in a **particular pre-determined context** and are **not adaptative regarding the amount of available resources** on the CPU nor frame duration in order to avoid drops but fixed optimizations considered during the production of the game in order to limit computations. Algorithmic optimizations are also developed to reduce the computations as much as possible but this is of course limited depending of the algorithm utilized to compute these tasks.

As we are unable to have global management of quality level for such tasks, it induces also the impossibility to adjust task duration depending on the platform the game is running on. Since a video game is supposed to be playable on a wide range of devices and in order to guarantee it will run respecting an expected frame rate, one solution at the moment is to **limit engine's computations to fit the minimal configuration**, thus limiting the amount of simultaneous AI, Explosion and physics computation to avoid overloads.
Another method to reduce frame duration and avoid frame drop is to carefully handle **scheduling of the engine**. In order to do so, engine developers use **static priority values (Low, Medium, High)** to prioritize the different tasks and a back and forth between game developers and engine programmers is set to determine if the engine can handle required computations, **determine critical path and prioritize the different tasks composing it**. As a matter of fact, the graphic loop is most of the time considered as the critical path in order to provide data to the GPU as fast as possible.
Unfortunately, video games being an interactive soft real time 3D simulation, the amount of **computations may strongly vary** depending on the different phases of the engine. For instance, when handling a great amount of NPC simultaneously, prioritizing AI treatment is preferable whereas physics intensive computations should be prioritized when explosion chain reaction occurs.
In this context **static prioritization of a single critical path cannot provide performances enhancement for the entire game** and a dynamic scheduling should be set up to adapt scheduling depending on the different phases of the video game. One challenge with dynamic scheduling is its **overhead**, which can present multiple issues in the context of video games. First, the moments when games need to adjust their scheduling are often the same moments they have the least time to spare. Additionally, given the limited resources in terms of memory and computation time that can be allocated to scheduling, a method with a **constrained memory and overhead** would be most suitable. Lastly, dynamic scheduling is often considered at risks for video games, as **dependencies aren't as controlled** as it should be, prioritizing tasks differently may induce data concurrency.

All these limitations induce a co-dependence between the game and the worse performing device.

### 1.3.3 Contributions

In this manuscript, we will detail the different contributions of the PHD including definition and **formalization of the Model of task scheduling for video games**. **Creation of Metrics** in order to quantify performances. **Implementation of a simulation** allowing to generate representative Frames or replay real ones extracted from the engine and estimate scheduling performance and frames duration. Using this simulation, we **evaluate different online and offline scheduling techniques**. We also **provide a time and memory bounded dynamic scheduling heuristic** based on Monte Carlo Tree Search Exploration in order to adapt scheduling to the different phases of the game. Finally we introduce a dynamic task skipping mechanism allowing **adaptative level of quality handling for targeted CPU tasks** based on the amount of resources available and estimated duration of incoming frames to avoid frame drop as well as it's evaluation on different platform for two recent AAA games from Ubisoft.

# Chapter 2

# Studying scheduling for video games

Very little literature exists about video game scheduling in academic field. In this context state of the art list heuristics have not been explored. Our goal in a first time is to provide extensive study regarding the capacity of classic High Performance Computing techniques to enhance performance of the engine. For this purpose, we provide a background on task scheduling, we introduce a simulator enabling to simulate and replay representative frame of a recent video game from Ubisoft. We then evaluate 11 different list scheduling heuristics on specific metrics for this problem and propose several structural optimisations for game engines. The following chapter is greatly inspired by the article published in Euro-Par2022 [34]

## 2.1 Background and Related work

To study the impact of list scheduling on game engine, we first need to define few aspects of the scheduling problem.

### 2.1.1 Task programming

In the context of this PHD, we consider a code paradigm used for decades in high-performance computing. Task programming is a method for structuring parallel programming in which complex computations are broken down into smaller, independent units of work called tasks. These tasks can be executed concurrently on multiple processors or cores, allowing for efficient use of resources and improved performance.

In order to orchestrate and organize the different tasks and their executions, developers define dependencies that describes the order in which tasks must be executed based on their input and output requirements. Dependencies is declared when the output of one task is required as input for another task. These tasks and dependencies are represented in a Directed Acyclic Graph (DAG). In a DAG each node in the graph represents a task, and directed edges represent dependencies between tasks. The edges indicate the direction of the dependency, pointing from the task producing the output to the task requiring it as input. The graph is acyclic, meaning it contains no cycles, which prevents circular dependencies and ensures that there is a valid execution order for all tasks.
To manage these dependencies, a scheduler is employed to coordinate task execution, ensuring that tasks are only started when their dependencies are satisfied.

### 2.1.2 Scheduling

Scheduling is the process of organizing and allocating tasks to various processing units (such as processors, cores, or threads) to efficiently execute a set of tasks concurrently. The goal of scheduling is to optimize resources utilizations, minimize execution time, and maintain load balance among available processing units while respecting task dependencies and constraints. It can significantly impact the efficiency, performance, and scalability of a system by minimizing execution time and, reducing communication overhead.

When scheduling game engine, we consider mainly List Scheduling. It involves assigning priorities to tasks based on certain criteria (e.g., task duration, critical path, DAG structure) and then allocating tasks to available resources in descending order of priority. This approach allows for flexible and adaptive scheduling, balancing workloads among resources, and providing reasonable performance in various parallel computing scenarios. However, its effectiveness depends on the choice of priority function and the specific characteristics of the task set and system resources. Despite its limitations, List Scheduling has proven to be a useful and efficient scheduling technique in many parallel computing applications, providing reasonable performance and adaptability to varying workloads and system conditions.

In the context of list scheduling, and to provide a better understanding of the method, we can split the scheduling method in two different matters.

The first one is the prioritization also called labeling. It consist in assigning priorities to the tasks composing the graph as shown on **Fig. 2.1**, these priorities will then serves as guidance for the second matter which is distribution. Changing the priority, will induce a change in the order of computation as seen in **Fig. 2.2**.

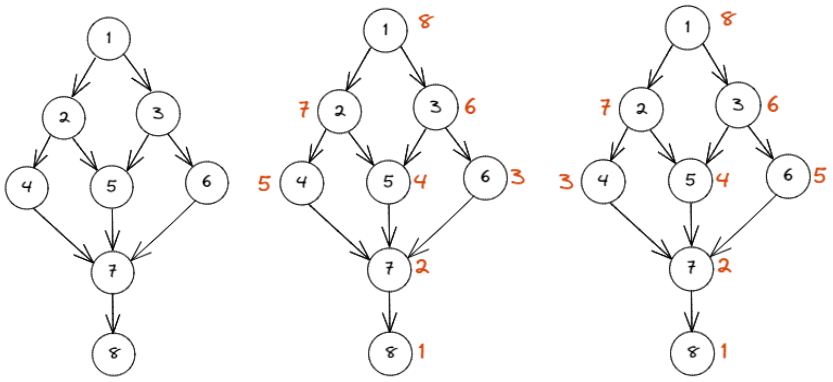



Figure 2.1: **Priority labeling examples (b) & (c) on task graph (a), priorities are written in red**

We denote two categories of prioritization. Primary the static labeling, consisting in setting priorities for the different tasks depending on information relative to the structure of the graph or overall tasks property for instance its mean execution time. Once these priorities are set, they cannot be updated during the execution. Even though this method is not optimal when variations in the amount of computations occur during execution, it has proven to be a useful heuristic when a clear critical path is identified in a quite stable environment. Moreover, it induces very little overhead as it does not require any further computations once all priorities have been set.

When the amount of computation vary during the execution modifying the critical path in a unpredictable way, Dynamic labeling is most often used. It consists in updating

the different tasks priorities along the execution depending on various information such as tasks processing time, waits etc. This method can provide increased performances depending on the problem but also induces an overhead, that may directly impact the quality of the execution depending on the chosen heuristic.

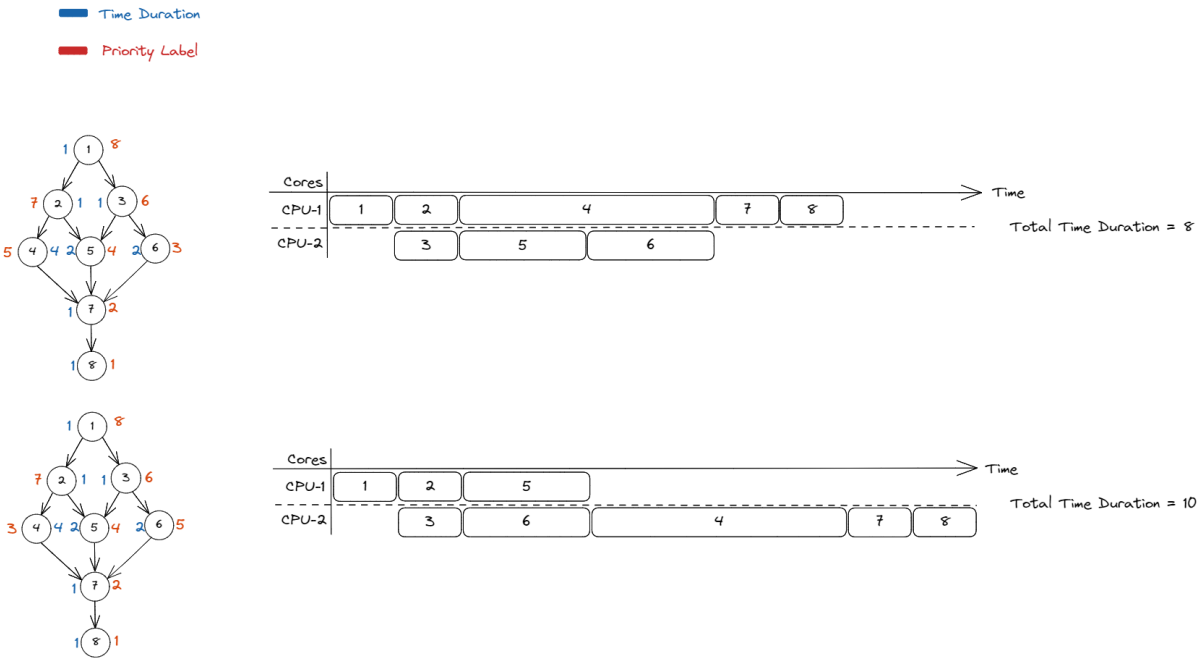



Figure 2.2: **Priority Labeling impact on the scheduling based on task graph from Fig.2.1 (a)**

The second matter we need to address regarding scheduling is the distribution. It consists in dispatching ready tasks among available computations resources. We will only consider CPU in this PhD because GPU computations are handled in a different way, making it difficult to apply classic scheduling techniques. There are two distinct methods to dispatch these tasks.

On the first hand, Lazy distribution that delays task assignment decisions to respect strictly the priorities set during the labeling phase. This approach aims to minimize overhead and maximize flexibility by allowing the scheduler to make allocation decisions based on the most up-to-date information about tasks dependencies, resource availability, and system state, resulting in better adaptability to dynamic changes in workloads and resource conditions.

On the other hand, eager distribution that proactively assigns tasks to resources as soon as they become available and their dependencies are satisfied. Following priority order only among available tasks at the moment. This approach emphasizes maximizing resource utilization and minimizing idle time, striving for efficient execution by keeping resources consistently occupied with tasks while respecting task dependencies and constraints. **Fig.2.3** Illustrates scheduling differences when choosing Lazy or Eager distribution for an execution based on a small graph. It is important to notice that none of these scheduling technics can be considered as a better option than another.

Video game are considered as a Soft Real Time system as described in **Sec.1.3**. Our scheduling problem has distinct characteristics that prevent traditional scheduling real-

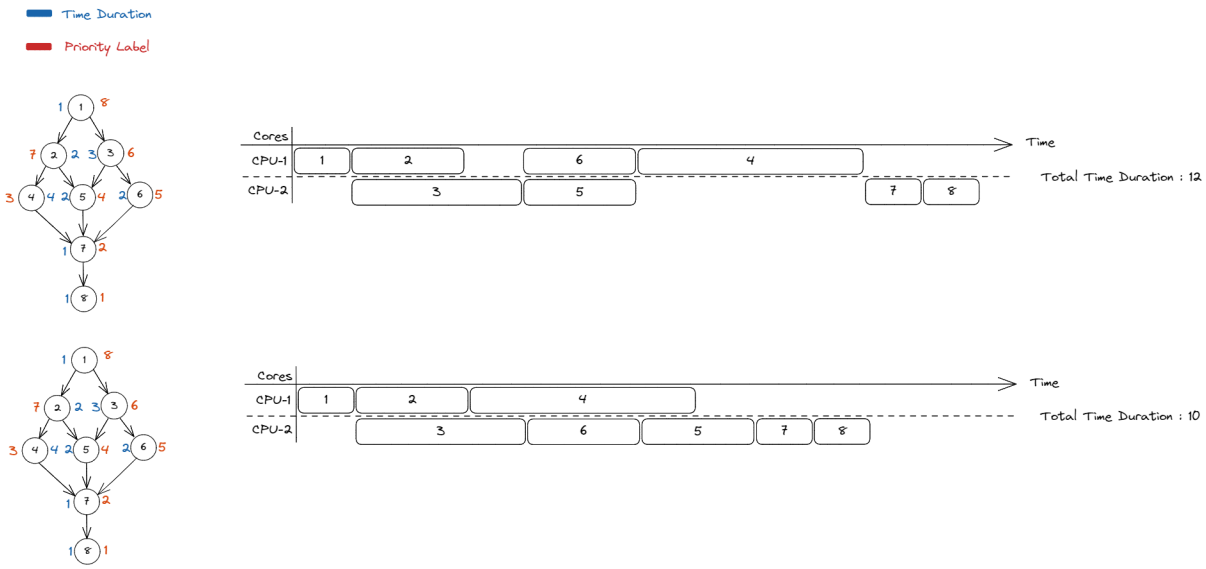



Figure 2.3: **Lazy Distribution (Top) and Eager Distribution (Bottom) impact on scheduling based on task graph from Fig.??(a)**

time or parallel tasks [28] techniques and heuristics from being used. Real-time scheduling most often considers independent, recurring tasks. Moreover Real Time and Soft real-time scheduling methods aim to maximize the number of tasks that meet their deadlines while maintaining overall system performance. Such is the case on the work of Nascimento and Lima [32], where earliest deadline first (EDF) heuristics are employed for scheduling soft and hard real-time tasks in parallel resources. EDF heuristic assigns priorities to tasks based on their deadlines, with tasks having earlier deadlines given higher priority. The scheduler then allocates tasks to resources based on these priorities, aiming to meet as many deadlines as possible. Unfortunately in the context of video games, the game engine contains dependent tasks with an entire task graph to be computed for each frame. Additionally the absence of individual deadlines for tasks induces the frame's end as a shared due date. These specificity's of the system prevent us from using EDF algorithm. An algorithm called DynFed was proposed to schedule parallel tasks with dependencies in real-time systems by Dai, Mohaqeqi, and Yi [18]. Nonetheless, it focuses on periodic, independent tasks whith parallel subtasks have dependencies, while our scheduling problem contains tasks with dependencies whose parallel subtasks are independent.

The video game industry also considered the issue of scheduling for video games. Major engine editors propose task creation and management system. For instance Unity provides its Job System [39] so engineers can write their own tasks with dependencies and let Unity schedule them. Unreal Engine 4 includes Tick Groups [40] to set when a task should be executed (e.g., before or after physics simulations). Recently, latest version of Unreal engine-5 provides the possibility to include TaskGraph [2] enabling to define dependencies between the tasks composing the gameplay. Godot on it's side introduces a system of "servers". A server is responsible for a specific area of the engine and is provided with a pre-determined amount of resources. Finally Intel proposed the Intel's Games Task Scheduler (GTS) [5], providing scheduling method for video games and introducing a multi level scheduler for tasks and substasks in order to enhance performances.

However even though all these different engine addressed the task management, none introduced scheduling techniques (Static or Dynamic) other than priority set by hand by developers using a limited set of values (Low, Medium, High).

Given the size of game engines, measurable in tens of Mo in code (for cpp and hpp files), their complexity and the impossibility to have stable test environment in order to evaluate the impact of different scheduling method. We decided to implement a scheduling simulation based on video game's task graph and profiling extracted from real execution allowing us an accurate task's representation in order to study representatives frames.

## 2.2 Extracting Data from Game Engines

To reconstruct an simulate the execution of the game engine. Several information are required. In the first place, we need to be able to identify and reconstruct the task graph structuring the execution with all its dependencies and the different tasks composing it. Then we need to be able to extract several characteristics value allowing to reconstruct tasks duration and amount of tasks, in order to reconstruct Frame. Finally we need to be able to generate a coherent and continuous execution as representative as possible of a real execution, using the Frames simulated.

### 2.2.1 Extracting Task Graph

As reminded earlier, Ubisoft's game engine studied is a software developed continuously for over a decade. In this context, the tasks have been implemented along the way and were not included in the first versions of the engine. This induces, a lack of structure to manage the tasks and their dependencies. The task graph representing the execution of a video game cannot be easily extracted, tasks are most of the time implemented directly in the code thus harder to track.

The graph presented in **Fig.2.4** was extracted by hand, using executions, tasks graph previously reconstructed (but not updated in years) and code study. We can notice the different areas handled by the engine, with 2 main different paths in the graph. On the left the path represents the graphical frame, in charge of transferring data to the GPU and enabled part of the computations on Image Buffer. On the right side, the path represents the engine frame, composed of physics, gameplay, characters etc.

Each task represents a functionality written by a given team in a given moment in the lifetime of the game engine and, no global design for the task graph as a whole. In this context, task interactions have to be kept simple. Internally, each task contains one or more independent, sequential subtasks following a fork-join model as illustrated in Fig. 2.5. The tasks composed of a set of subtasks are represented in red in the figure.

This specific structure of subtasks can be easily explained due to the nature of the computations and the code organization of a video Game. Usually video games is developed as a Entity Component System aiming to decouple the behavior and data from game objects. This structure allows developers to create complex and diverse game entities by composing them from reusable building blocks. For instance, a task such as "UpdateComponents" will be composed of $N$ subtasks, $N$ being the amount of Components constituting the game at a given point. Subtasks then execute the update of the corresponding components. In this context, Equation 2.1 defines the duration of task $t$ denoted $d_f(t)$ when composed of multiple subtasks. $\theta_f(t^i)$ represents the starting time of subtask $t^i$.

$$d_f(t) = \max_i(\theta_f(t^i) + d_f(t^i)) \tag{2.1}$$

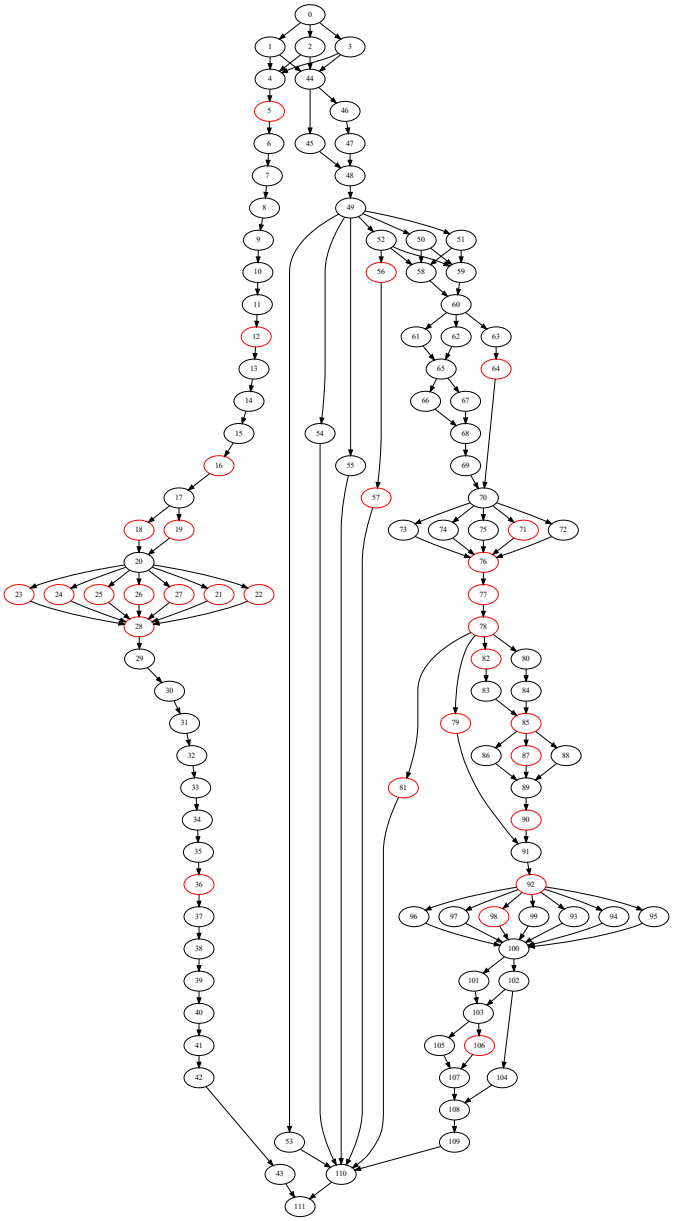


Figure 2.4: **A Directed Acyclic Graph representing game engine tasks (vertices) and their precedence constraints (edges, from top to bottom). Tasks in red are composed of multiple sequential subtasks.**

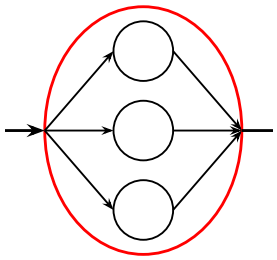


Figure 2.5: **Fork-join parallelism inside a task. Each small circle represents one sequential subtask.**

### 2.2.2 Extracting Characteristics Information through Profiling

To simulate the behavior of the engine, we propose to analyse task duration and amount of subtasks generated on different gameplay phases and on different maps. To study such behavior, we use profiling tools as seen in **Fig.2.6**. These tools enable to capture a continuous execution of the game, and provides detailed information and statistics regarding the different aspects its execution. Moreover these tools also provide a visual display of the execution highlighting the distribution of tasks on threads. This enables to quickly identify idle time in the scheduling.

To measure characteristics values of the game, we use two different "states" of the execution. Primary, when the game is running in a "smooth mode" (no NPC, no explosion to limit computations). Then when the game is running in an "overload mode" (maximum NPC, important amount of explosions and particles handling).
These two different captures of the game, repeated for a great amount of frames (more than 3000) and for several maps, allows us to identify the evolution of task duration and, amount of subtasks depending on the load variation of the engine.

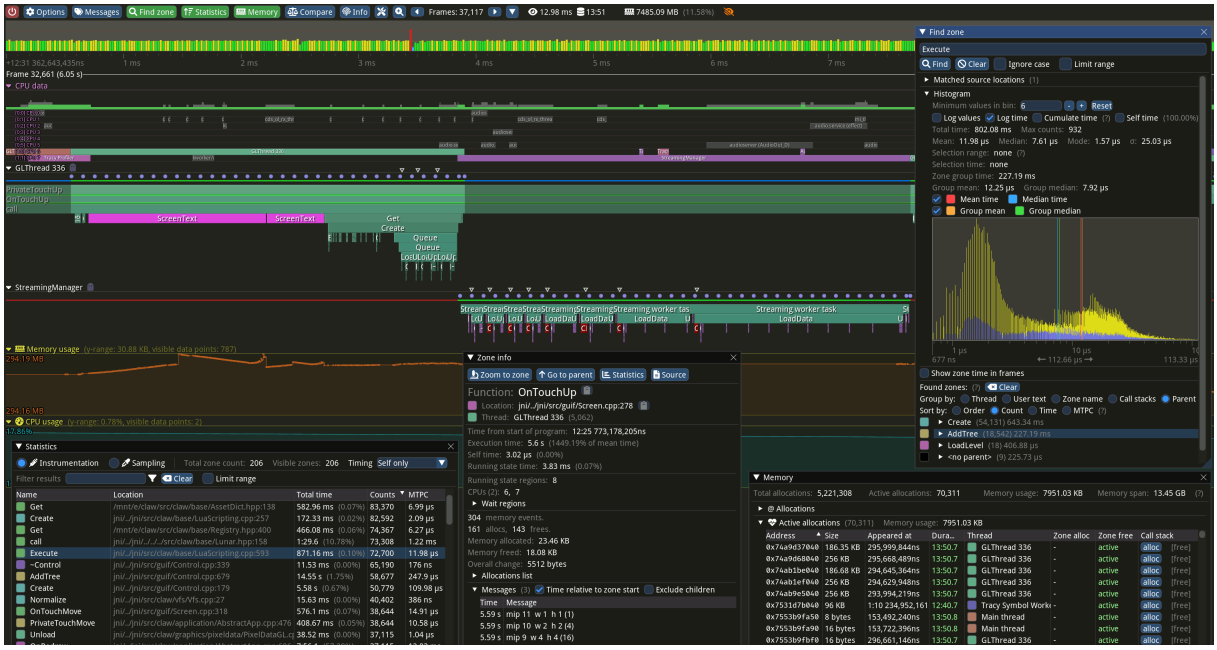


Figure 2.6: **Example of Tracy view, an open source frame profiling tool [44]**

We extracted several metrics, such as minimum, maximum, mean, and standard deviation for tasks duration. In the same way, we extracted the same metrics for subtasks as well as their number. These metrics were extracted under both flawless execution and overloaded conditions. They provide valuable data points allowing to simulate tasks and subtasks composing a frame.

## 2.3 Game Engine Simulation

At that point, using the different information extracted through profiling and task graph reconstruction, we are able to simulate and reconstruct representative frame. Even though these frames are not extracted from real executions, statistical analysis of such frame can provide insight on the performance provided through various scheduling techniques.

### 2.3.1 Tasks Simulation

On the first hand, we noticed tasks tend to follow a lognormal distribution rather than a normal one. This can be easily explained by the implication of multiple small independent factors affecting the duration of tasks.[30]. On the other hand, when the amount of subtasks grow, it tends to follow a linear evolution directly correlated with the level of engine overload.

Using these information, we define in Eq. 2.2 the processing time $p_j^{sub}$ of a subtask of task $j$ with overload $l \in [0, 1]$. This requires computing the minimum and maximum values for each task ($p_j^{min}(l)$ and $p_j^{max}(l)$, respectively), and sampling from a log-normal distribution with parameters $\mu_j(l)$ and $\sigma_j(l)$. These four values are computed by combining the measured values for the *low* and *high* states of the game engine in proportion to the overload, as represented in Eqs. 2.3, 2.4, 2.5, and 2.6.

$$p_j^{sub}(l) = \max p_j^{min}(l), \min p_j^{max}(l), LN(\mu_j(l), \sigma_j(l)) \tag{2.2}$$

$$p_j^{min}(l) = (1 - l) \cdot p_{j,low}^{min} + l \cdot p_{j,high}^{min} \tag{2.3}$$

$$p_j^{max}(l) = (1 - l) \cdot p_{j,low}^{max} + l \cdot p_{j,high}^{max} \tag{2.4}$$

$$\mu_j(l) = (1 - l) \cdot \mu_{j,low} + l \cdot \mu_{j,high} \tag{2.5}$$

$$\sigma_j(l) = (1 - l) \cdot \sigma_{j,low} + l \cdot \sigma_{j,high} \tag{2.6}$$

We compute the processing time $p_j(l)$ of task $j$ by adding together the processing times of its subtasks. This is represented in Eq. 2.7, where $s_j(l)$ represents the number of subtasks of task $j$ with overload $l$. This number is computed in a similar fashion to other overload-dependent parameters, as shown in Eq. 2.8. Finally, we can compute the processing time contribution of each task when computing the critical path in the graph by considering its slowest task. This is represented by function $c_j$ for task $j$ in Eq. 2.9.

$$p_j(l) = \sum_{k=1}^{s_j(l)} p_j^{sub}(l) \tag{2.7}$$

$$s_j(l) = \lceil (1 - l) \cdot s_{j,low} + l \cdot s_{j,high} \rceil \tag{2.8}$$

$$c_j(l) = \max_{k \in [1, s_j(l)]} p_j^{sub}(l) \tag{2.9}$$

### 2.3.2 Frame and Sequence Simulation

In order to reconstruct a representative frame of the engine using the Task Graph described in Sec.2.2.1. We define a load proportion (between 0 and 1), and generates all the different tasks and their duration, including subtasks. Nonetheless in order to simulate

the scheduling of the different tasks on the platform, we need to take into account the amount of resources available on the test platform.

Our goal in this simulation is not only to be able to reconstruct a frame but also to represent a full execution of the engine. In order to do so, we need to generate a set of continuous frame with small load variations in order to mimic a rise and fall in the load of the game depending of the player's action.

We provide a simplified description of the problem of scheduling game engine tasks for a single frame using Graham's notation [19]. The machine environment of our problem is composed of parallel and identical computing resources (namely, cores in a CPU). The job characteristics and scheduling constraints follow the task model provided in Section 2.3.1. In short, our tasks have precedents constraints, different processing times, and the same due date. Last, for the objective function, we define $C_j$ as the completion time of task $j$, and we define its lateness $L_j$ in Eq. 2.10 and its tardiness $T_j$ in Eq. 2.11, where $d_j$ represents the due date of the task.

$$L_j = C_j - d_j \tag{2.10}$$

$$T_j = \max(L_j, 0) \tag{2.11}$$

Given the aforementioned characteristics, this scheduling problem can be represented as $P|prec, d_j = d|T_{max}$, which is NP-Hard. Nevertheless, this representation does not capture all the details of our problem in practice, mainly due to imprecision on the processing times of tasks, and to potentially varying numbers of computing resources.

The processing times of tasks are assumed to be known in classic, deterministic scheduling algorithms. Yet, in practice, our tasks have stochastic processing times, and our scheduling algorithms are mostly dependent on measurements from previous frames to estimate their current behavior (e.g., how much overload to expect for the current frame). In this sense, using the notation $P_j$ to represent stochastic processing times [12, Chapter 1], our scheduling problem would be closer to $P|P_j, prec, d_j = d|T_{max}$. We have found no optimal algorithm for this stochastic scheduling problem.

Regarding the number of computing resources available, it may vary during execution time due to two main reasons. The first reason is that video games run on personal computers, so their operating system may take possession of some cores from time to time to run its own tasks or other programs. The second reason is that the game engine itself has other sporadic tasks that do not belong in the usual task graph. For instance, these tasks can be responsible for prefetching data from disk, which does not happen every frame. In practice, this has the same effect of removing computing resources from the pool available for scheduling the task graph. We study the effects of having different numbers of resources later in Section 2.5.

The quality of a scheduling solution for multiple frames is based on its results for each frame (namely, its maximum tardiness). Consider the total number of frames $F$ and a given frame $f \in [1, F]$. We denote the maximum tardiness of frame $f$ as $T^f$. Using this information, we define three possible optimization metrics to minimize, namely the Slowest Frame ($SF$), the number of Delayed Frames ($DF$), and the Cumulative Slowdown ($CS$). They are presented in Eqs. 2.12, 2.13, and 2.14, respectively. The Slowest Frame

represents the moment with the worst frame rate to be noticed by a player. The number of Delayed Frames quantifies the periods of reduced frame rate that can be noticed. Lastly, the Cumulative Slowdown qualifies these periods. Using these three metrics, we can compare different scheduling algorithms for game engines.

$$SF = \max_{f \in [1,F]} T^f \tag{2.12}$$

$$DF = \sum_{f \in [1,F] \wedge T^f > 0} 1 \tag{2.13}$$

$$CS = \sum_{f \in [1,F] \wedge T^f > 0} T^f - d \tag{2.14}$$

## 2.4 Scheduling Strategies for Game Engines

Eager scheduling heuristics, most of the time based on the possibility to estimate task's length have proven great efficiency, however, performances can be consider inconsistent for a given heuristic depending on graph's structure, granularity and communication cost. In practice, the complexity of such algorithms extends from O(n.log(n)) [9] to O(n4) which are impracticable for graphs considering hundred's of millions of tasks. However in our current study, the engine is performing a graph composed of few hundreds tasks every 16.6ms, this relatively low amount of tasks allows us to execute such heuristics, moreover engine redundancy gives us a high precision estimation for tasks duration. It was shown [8] [13] that accurate estimation of tasks duration is not essential. While considering only slight disturbance in the models, dynamic heuristics are able to cope with such variations and only significant perturbations are likely to have real impacts regarding performances degradation.

### 2.4.1 Scheduling strategies

Given the complexity of our scheduling problem and the absence of an algorithm focused on solving it, we have selected — and, sometimes, adapted — several scheduling algorithms from the state of the art for our experiments. All of these algorithms follow a list scheduling : whenever a resource becomes available, it takes the task with the highest priority and executes one of its subtasks. Besides its simplicity, list scheduling is also attractive due to the way it can adapt to changes in the number resources available, and by its known bound of $2-\frac{1}{m}$ to an optimal solution for the problem $P|prec|C_{max}$ ($m$ being the number of resources). Furthermore, all of our list scheduling algorithms respect an additional rule from the game engine: graphics tasks are only executed in resource 0, and they maintain a higher priority than other kinds of tasks. Moreover, the scheduling of all the different tasks follows an eager distribution as introduced in Sec.2.1.2. We present the chosen algorithms in Table 2.1. We classify them by the way they compute the priority of each task.

*Local* algorithms use only information from the task to compute its priority, which leads to a lower complexity or overhead. The opposite are *global* algorithms that tend to consider the paths in the task graph. *Online* algorithms require information obtained at run time, while *offline* algorithms can pre-compute task priorities. Finally, *time*-aware algorithms use timing information to compute priorities (mainly the processing time of the

tasks). These times may refer to information capture in the last frame, or from previously profiled and averaged processing times. This method to use the previous frame as an oracle is acceptable because video game is a continuous interactive 3D simulation. The amount of computations vary little from one frame to another during classic execution. Because of the interval between two frames (16.6ms) the state of the game cannot evolve drastically. The amount of ennemies will remain roughly stable, as well as the position of the player. This leads to only minor modifications in the amount of computations required on one frame to the next. However we are aware that this approximation may be false in some cases. For instance in the case of multiple initialisation of NPC during the loading of a world to populate it. But we consider this as ponctual events, causing little disturbance of the engine behavior.

Table 2.1: Characterization of tested scheduling algorithms.

| Acronym | Meaning | Scale of information | Processing | Time-awareness |
|---|---|---|---|---|
| FIFO | First In, First Out | — | — | — |
| LPT | Longest Processing Time first | Local | Online | Previous frame |
| SPT | Shortest Processing Time first | Local | Online | Previous frame |
| SLPT | LPT at a subtask level | Local | Online | Previous frame |
| SSPT | SPT at a subtask level | Local | Online | Previous frame |
| HRRN | Highest Response Ratio Next | Local | Online | Previous and current frame |
| WT | Longest Waiting Time first | Local | Online | Previous and current frame |
| HLF | Highest Level First | Global | Offline | — |
| HLFET | HLF with Estimated Times | Global | Offline | Mean |
| CG | Coffman-Graham's algorithm | Global | Offline | — |
| DCP | Dynamic Critical Path | Global | Online | Previous frame |

The First In, First Out scheduler serves as our baseline. It represents the original implementation in the game engine, which makes the simplest decisions and avoids overhead.

This heuristic was chosen to represent the scheduling of a video game because of the overuse of high priority that is likely to happen in video game engine. Assigning too many tasks to the "high" priority level can diminish the value of the priority system. When a large number of tasks share the same priority, the scheduler is forced to treat them equally, which may result in a de facto FIFO execution order. This is caused most of the time as developers implementing tasks are also the ones defining their level of priority. We noticed a tend to assign in doubt high priority to tasks as it guarantee fewer executions issues (a priority set to low may induce a delay that generate issues if data dependencies are not well known).

Next, we have online algorithms whose task priorities are based on time-related information. LPT [20] gives the highest priority to the tasks with the available task with the highest processing time $p_j$ in the last frame. This strategy is applied in many contexts as it improves the bound to the optimal solution of standard list scheduling. Meanwhile, SPT [23, 10] takes the opposite approach, which leads to an optimal solution to the problem $R||\sum C_j$. SLPT (and SSPT) follows the same logic of LPT (SPT), but instead does it at a subtask level (i.e., using $p^{sub}_{j,k}$, which represents the processing time of the $k$th subtask of $j$ in the previous frame).

Instead of employing processing times to compute priorities, HRRN and WT use information related to the moments a task becomes available in the priority queue in the current frame (its *ready time* $r_j$), its first subtask started executing (its *beginning time* $b_j$),

and its last subtask finished executing (its *completion time* $C_j$). HRRN uses these values to compute a *response ratio* for the priorities as $\frac{C_j - b_j}{C_j - r_j}$. WT, in its turn, computes the difference between the moment a task becomes ready and the moment it starts executing ($b_j - r_j$). In both cases, tasks with smaller values are given a higher priority.

Other algorithms employ information about the task graph offline to prioritize tasks that may delay the completion of the last task (the *exit node*). HLF [24] provides an optimal solution to the scheduling problem $P|intree, p = 1|C_{max}$ by prioritizing tasks in the longest paths to the exit node (thus optimizing for the critical path of unitary tasks). HLFET [4] extends the basic strategy of HLF by considering estimated processing times. In our case, these estimations are based on the mean processing times captured by our task model. CG [15] uses the labeling algorithm developed by Coffman and Graham, which has been shown to be optimal for the problem $P2|p_j = p, prec|C_{max}$.

Our final algorithm, named DCP, combines global information online. This gives DCP the highest amount of information, with the potential drawback of a higher overhead. It computes the priority of a task in two ways. If task $j$ is identified as part of the critical path in the previous frame, it is added to the head of the priority queue. Else, task $j$ is added to the queue with priority $prio(j)$ after all tasks in the critical path. $prio(j)$ is computed in Eq. 2.15 using information from the previous frame, the set of successors of task $j$ in the graph as $succ(j)$, and the number of resources $m$. In short, the priority of a task is computed based on the highest priority among its successors and the maximum between an estimation of its parallel execution and its slowest subtask.

$$prio(j) = \max_{i \in succ(j)} prio(i) + \max\left(\frac{\sum_{k=1}^{s_j} p_{j,k}^{sub}}{\min(m, s_j)}, \max_{k \in [1, s_j]} p_{j,k}^{sub}\right) \tag{2.15}$$

## 2.5 Experimental Evaluation

To evaluate all different lists scheduling introduced in the previous section, several experiments were performed in a controlled environment.

### 2.5.1 Implementation and Methodology

Our experiments employ an in-house scheduling simulator written in C++ and based on a modern game engine introduced in Sec.2.3.2. Given a complete description of the task graph, the number of frames to simulate, the number of resources, a scheduling algorithm, and a random number generator (RNG) seed, it deterministically simulates the scheduling and execution of all tasks. Its determinism is important to enable direct comparisons between scheduling algorithms and experimental scenarios. The simulation represents a perfect scheduling environment with no overhead from the scheduling algorithm, data locality, or other sources of interference. Although less realistic than testing in the actual game engine, we believe our simulation is less prone to noise, making it easier to understand and compare results.

Each of our simulations runs for 200 frames with variations to the overload parameter ($l$). Overload starts at zero in the first frame, and it increases linearly until achieving its maximum value (one) in the $101^{st}$ frame. It then starts decreasing in the same rate (0.01 per frame), until arriving to the value of 0.01 for the last simulated frame.

All parameters required to model the tasks (Eqs. 2.2 to 2.6 were obtained in a separate, dedicated machine used for video game development with a Intel Xeon E5-1650V4 / 3.6

GHz (4 GHz) (6 cores) 15Mo cache.

For each scenario and scheduling strategy, we ran simulations using from 4 up to 20 resources (our standard case is considered to be 12 resources, as this is a common number of cores in current gaming processors). Among these resources, one is dedicated to graphics tasks, meaning it only takes other kinds of tasks when no more graphics tasks are available. In each situation, we varied the RNG seed in the interval $[1, 50]$, leading to a sample size of 50 for each scheduler, number of resources, and scenario. In total, this represents the simulation of $200 \times 50 \times 17 \times 11 \times 3 = 5,610,000$ frames (excluding the simulations for the Critical Path).

For the statistical evaluation of our experiments, we first employed descriptive methods (sample statistics and plots) to understand our results and to verify that no errors or outliers were present. We then followed with inferential methods. Setting our tests to a 5% significance level, we used Kolmogorov-Smirnov (KS) tests to check if samples came from normal distributions for the three metrics (SF, DF, and CS) whenever relevant. In all tested cases, we could not reject the null hypothesis that the results came from a normal distribution (all p-values $> 0.05$). We then ran F-tests to compare the variances of relevant pairs of samples. Again, in all cases, we could not reject the null hypothesis that the samples had the same variance. Given these statistical results, we used Student's T-test for all relevant comparisons discussed in the next sections.

All of the simulation results where obtained on a Dell Latitude 7420 notebook with an Intel Core i7-1185G7 processor, 32GB of LPDDR4 RAM (3200MHz), and an Intel PCIe SSD with 512GB of capacity. The machine ran on Ubuntu 20.04.3 LTS (kernel version 5.13.0-1022-oem), and g++ 9.3.0 was used for the compilation of the simulator (-O3 flag). Our experiments were replicated on a second machine with different software and hardware to verify that their results were deterministic. The results were plotted and analyzed using Python 3.8.10 with modules matplotlib (3.3.1), pandas (1.3.2), seaborn (0.11.2), and scipy (1.7.1).

### 2.5.2 Discussion and Results

We summarize the main performance results when scheduling tasks on 12 resources in Table 2.2 and Fig. 2.7 (the smallest the values, the better). Table 2.2 shows the values of Slowest Frame, Delayed Frames, and Cumulative Slowdown (rows) computed for each scheduling strategy (columns). These values represent the averages over 50 executions. The first column presents FIFO (our baseline) and the last column shows the values for the Critical Path, which represents the optimal results with unlimited resources. Meanwhile, the general distribution of values for the different metrics is illustrated in boxplots in Fig. 2.7.

| | FIFO | LPT | SPT | SLPT | SSPT | HRRN | WT | HLF | HLFET | CG | DCP | Critical Path |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SF (ms) | 32.88 | 32.87 | 32.40 | 32.37 | 32.78 | 32.37 | 32.39 | 32.48 | 32.54 | 32.38 | 32.38 | 28.37 |
| DF (frames) | 72.48 | 72.86 | 68.82 | 68.7 | 72.02 | 68.50 | 68.74 | 69.52 | 69.98 | 68.74 | 68.70 | 45.50 |
| CS (ms) | 375.30 | 376.83 | 344.37 | 343.12 | 370.86 | 342.99 | 342.52 | 349.52 | 353.69 | 343.17 | 342.87 | 171.40 |

Table 2.2: **Average metrics for all scheduling strategies on 12 resources.**

At first, we can notice that the smallest performance improvements were achieved on the SF metric. This indicates that, under the worst load conditions, no scheduling strategy was able to avoid the large increases in frame duration. Nonetheless, even if the SF improvements achieved by strategies such as WT and CG were minor (a factor of


Figure 2.7: **Boxplots of the different metrics on 4, 8, 12, 16, 20 resources. (Vertical axes start at different points to emphasize differences.)**

about 1.015), they were still statistically significant (p-values = $5.12 \times 10^{-33}$ and $9.64 \times 10^{-30}$, respectively). This was not the case for LPT (p-value = 0.69). Still, we can see that the Critical Path for these simulations was, on average, equal to 28.37 ms, which is only better than FIFO by a factor of 1.159, and still 1.70 larger than the desired frame duration (16.667 ms).

Improvements brought by the different scheduling strategies are much more noticeable when we consider the DF and CS metrics. These improvements are present for strategies both local and global, online and offline (Table 2.1). For instance, WT (local, online) reduced the delayed frames by a factor of 1.054 over the baseline (p-value = $4.15 \times 10^{-21}$), as did CG (global, offline) (p-value = $5.65 \times 10^{-21}$). DCP (global, online) did the same by a factor of 1.055 (p-value = $4.21 \times 10^{-21}$). Interestingly enough, we cannot say that these three strategies perform differently using the DF metric (all p-values $> 0.05$), but we can do so using the CS metric (all p-values $< 0.05$) — even though their boxplots in Fig. 2.7 look very similar.

In order to better understand the effects of the scheduling strategies on the duration of the frames, we show in Fig. 2.8 histograms showing the frame duration reductions achieved by LPT, WT, CG, and DCP. These values were computed by subtracting the duration of each frame scheduled by FIFO by the respective value for each algorithm. These subtractions are done for each pair of frame number and RNG seed. The horizontal axis is organized in bins of $20\mu s$ truncated in a range of $-1000\mu s$ to $1000\mu s$ [1], while the vertical axis shows the frequency that frame duration reductions fall into each bin. A positive reduction means that the scheduling strategy reduces the duration of a specific frame, thus improving performance.

Three relevant aspects can be noticed in this figure. First, we can notice that a strategy such as LPT (Fig. 2.8 has most of its frame duration reductions centered around $0\mu s$, indicating that its decisions lead to schedules very similar to FIFO. Second, the other illustrated strategies have results mostly centered around $500\mu s$, but they also show slightly different curves. This shows that, although they make different scheduling decisions with varied effects on the duration of each frame, they are still able to improve the performance of the game engine in their own ways. Third, all scheduling strategies show values that are below $0\mu s$, demonstrating that no single algorithm is able to always improve performance.

Performance improvements by some scheduling strategies can also be seen across different numbers of resources. This is illustrated in Fig. 2.9, where the average DF of selected schedulers (vertical axis) is shown from 4 up to 20 resources (horizontal axis). In general, we can see that FIFO and LPT perform similarly, as do WT, CG, and DCP among themselves with 6 or more resources. We can also notice that the absolute difference in number of delayed frames between FIFO and other strategies tends to decrease with increasing number of resources. In our situation, this difference goes from about 7 frames on 4 resources down to under 2 frames on 20 resources. In a sense, this is to be expected, because it becomes harder to saturate resources with tasks as the number of resources increases, which in turn reduces the delay seen on important tasks from the critical path.

The logarithmic shape of the curves in Fig. 2.9 underscores the diminishing returns of larger numbers of resources. Besides, even when scheduling tasks on 20 resources, we can still notice a gap of about 10 frames between the DF of the best schedulers and the Critical Path.

To better illustrate this difference, we contrast the frames duration of the Critical Path

[1] Some frame duration changes fall outside the illustrated range.

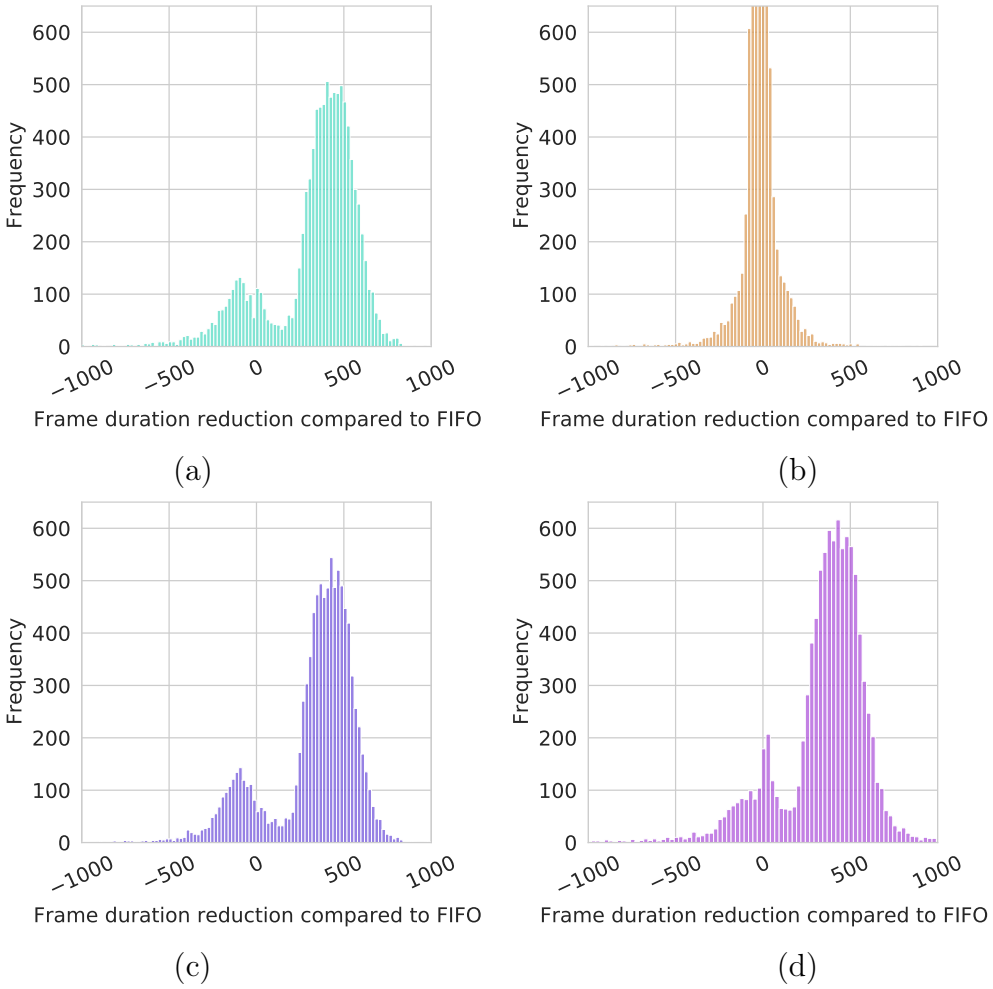



Figure 2.8: **Histograms presenting frame duration reductions in µs compared to FIFO. For WT(a), LPT(b), CG(c) and DCP(d) (Positive values mean that the scheduler provides a shorter frame duration.)**

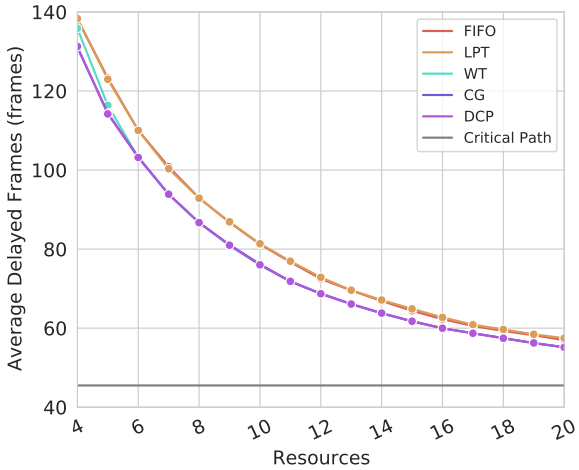



Figure 2.9: **Average number of delayed frames for a subset of the schedulers on different number of resources. The vertical axis starts at 40 frames to emphasize differences.**

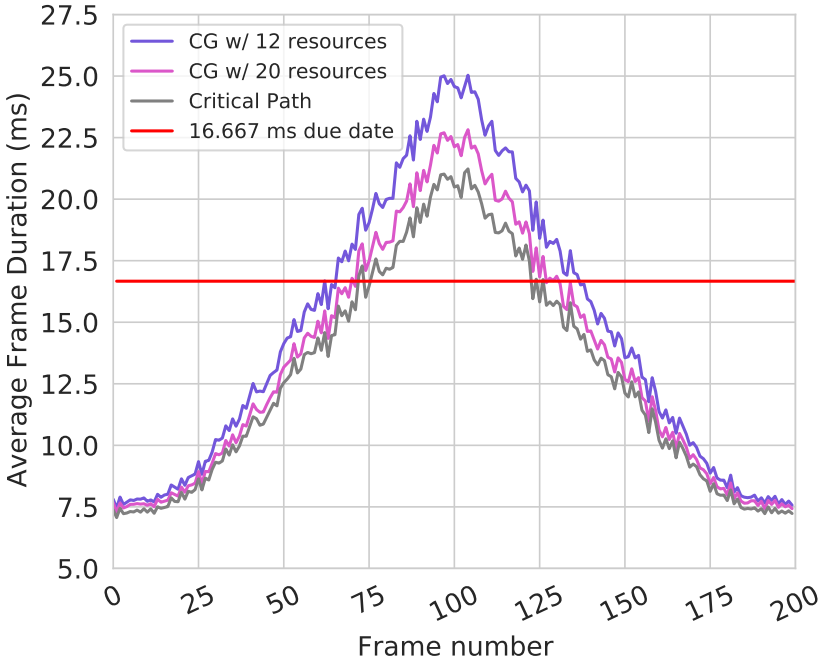



Figure 2.10: **Average duration of each frame for CG on 12 and 20 resources, and for the Critical Path.**

and CG in Fig. 2.10. In it, the horizontal axis represents the simulated frames in order, while the vertical axis represents the average duration of the frame for CG with 12 and 20 resources, and for the Critical Path. The red horizontal line represents the 16.667 $ms$ due date to respect to achieve 60 fps.

Fig. 2.10 exposes the change in frame duration following the change in overload (which peaks around frame 100). While the Critical Path starts surpassing the due date with a overload around 0.75, CG has the same issue for even smaller lags depending on the number of resources. Although the increase from 12 to 20 resources was able to reduce the gap between the achieved frame duration and the optimal one from $4ms$ down to $2ms$ for the slowest frames, we were surprised that such gap still remained. This motivated the changes presented in the next scenario.

## 2.6 Changing the Video Game Scheduler - Multi level approach

In light of the gains achieved by the different scheduling strategies and their limitations, we moved our attention to internal list scheduling mechanism available in the game engine. We proposed to change the order that the subtasks are chosen for execution. Originally, the scheduler takes the first non-executed subtask from the highest priority task available. We have instead chosen to sort the subtasks in a task by non-increasing order of execution time. This choice is motivated by the profiling observations of real executions where the subtasks are chosen in a first in first out method and results in idle time on the cores while waiting for the last ones to be finished. This lack of load balancing and management on these very particular subtasks was inducing great wait time and thus a significative increase in the duration of the frame. Considering the sorting of subtask is feasible in a real game engine because it does not affect the actual execution of the tasks nor does it

affect the dependencies in the task graph. Additionally, the developers of a video game can provide clues of the most important (in term of computational resources) subtasks composing a task statically or based on simple internal parameters. For instance, the character handled by the player will most of the time be the one with the greatest amount of detail as it is the closest from the camera as well as the most animated of the game. In the same manner, NPC's with the highest level of detail will be closest from the player and are more likely to have interactions with him thus inducing more computations for a given frame. In this context, prioritizing the updates concerning the main character as well as close range NPC induces prioritizing subtasks with the greatest amount of computations. Moreover, the continuous execution of the game can also give us some hints on the upcoming execution where tasks bounded to a NPC with highest computation time will be more likely to have a similar behavior on the next frame. Even though this method is not a perfect oracle, this method provides us rough size classification of the subtasks.

### 2.6.1 Discussion and Results

The performance results achieved in this scenario using 12 resources are summarized in Table 2.3. It follows the same organization of Table 2.2 with the addition of rows the percentage decrease in the different metrics when compared to the results of the previous table. The improvements brought by sorting the subtask is clearly noticeable for all scheduling strategies and metrics.

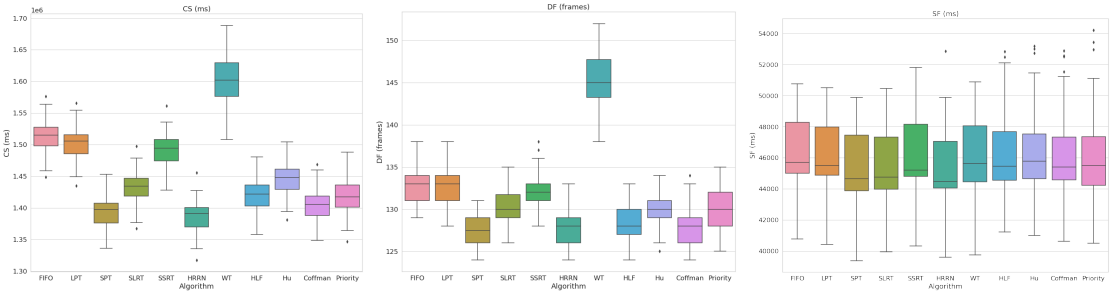
Meanwhile, the general distribution of values for the different metrics is illustrated in boxplots in Fig. 2.11. To provide more information and result comparisond of scheduling methods with sorted subtasks, we propose Fig.2.13 and, Fig.2.13 representing in the same way as Fig. 2.8 histograms showing the frame duration reductions achieved by LPT, WT, CG, and DCP.
We also provide in Fig.2.17 and, Fig.2.19 a comparison between Multi level scheduling and Baseline scheduling for heuristics LPT, WT, CG, and DCP following the same method.

| | FIFO | LPT | SPT | SLPT | SSPT | HRRN | WT | HLF | HLFET | CG | DCP | Critical Path |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SF (ms) | 29.29 | 29.25 | 28.67 | 28.96 | 29.15 | 28.65 | 28.63 | 28.74 | 28.80 | 28.63 | 28.64 | 28.37 |
| *(% change)* | -10.93 | -11.01 | -11.52 | -10.53 | -11.08 | -11.49 | -11.60 | -11.51 | -11.49 | -11.57 | -11.57 | - |
| DF (frames) | 54.32 | 54.28 | 49.52 | 51.88 | 53.62 | 49.10 | 48.98 | 50.12 | 51.08 | 49.38 | 49.34 | 45.50 |
| *(% change)* | -25.06 | -25.50 | -28.04 | -24.48 | -25.55 | -28.32 | -28.75 | -27.91 | -27.00 | -28.16 | -28.18 | - |
| CS (ms) | 217.91 | 217.32 | 189.36 | 203.24 | 212.60 | 187.68 | 186.62 | 192.86 | 197.13 | 187.81 | 187.84 | 171.40 |
| *(% change)* | -41.94 | -42.33 | -45.01 | -40.77 | -42.67 | -45.28 | -45.52 | -44.82 | -44.27 | -45.27 | -45.22 | - |

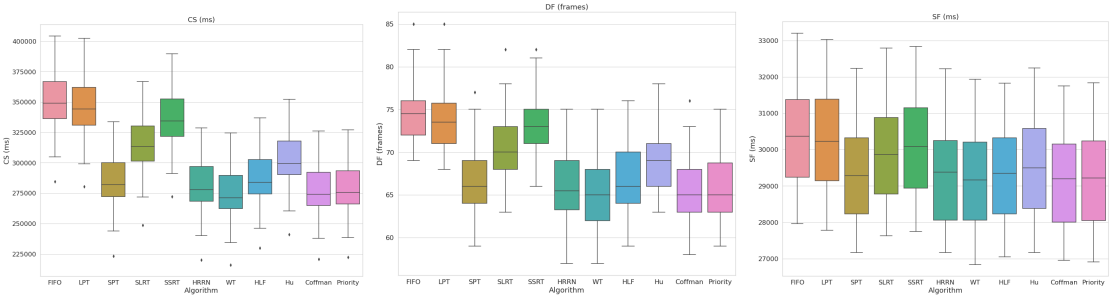
Table 2.3: **Average metrics for all schedulers over 12 resources with sorted subtasks. Percentage reductions are calculated in comparison to Table 2.2.**

For instance, the change to the average value of SF for FIFO from $32.88ms$ to $29.29ms$ represents a 10.93% decrease in time (or an equivalent reduction factor of 1.123), which is greater than the benefits we were able to achieve by changing the scheduling strategy only. Still, in many cases, the improvements were even better for some scheduling strategies than for FIFO, leading to greater cumulative improvements. We notice on Fig.2.11 an improvement for CS, DF and, SF for every amount of resources

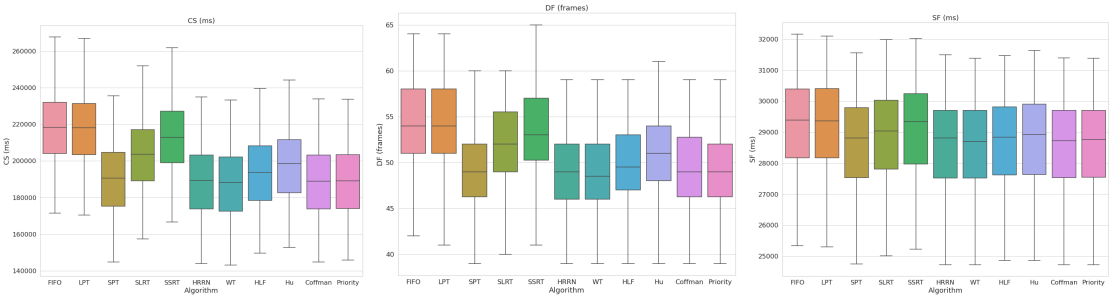
We can observe that schedulers illustrated in Fig. 2.13 Fig. 2.15 (namely, WT, CG, LPT and, DCP) show trends that are similar to the results in Scenario I. We notice LPT tend to show little to no improvement, while all the others performs better with a small



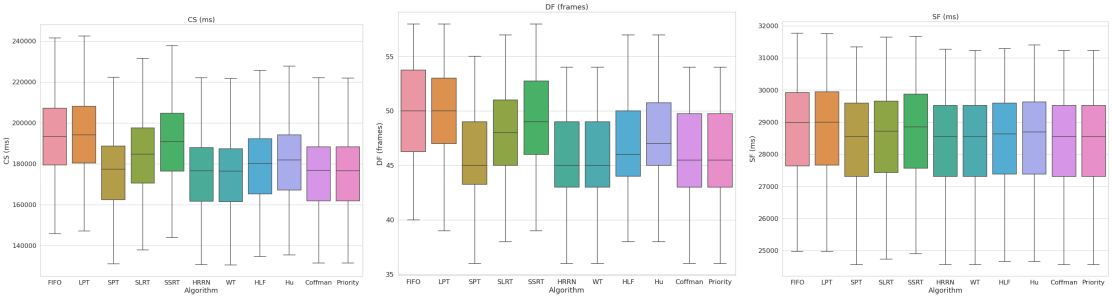

(a) 4 cores



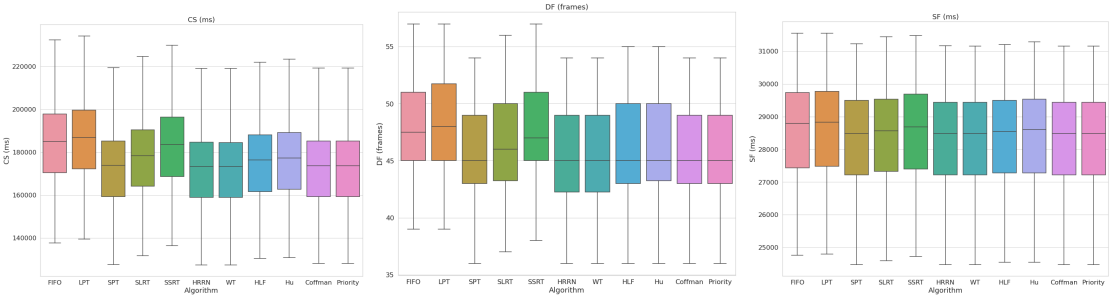

(b) 8 cores




(c) 12 cores




(d) 16 cores




(e) 20 cores

Figure 2.11: **Boxplots of the different metrics on 4, 8, 12, 16, 20 resources with multi-level scheduler. (Vertical axes start at different points to emphasize differences.)**

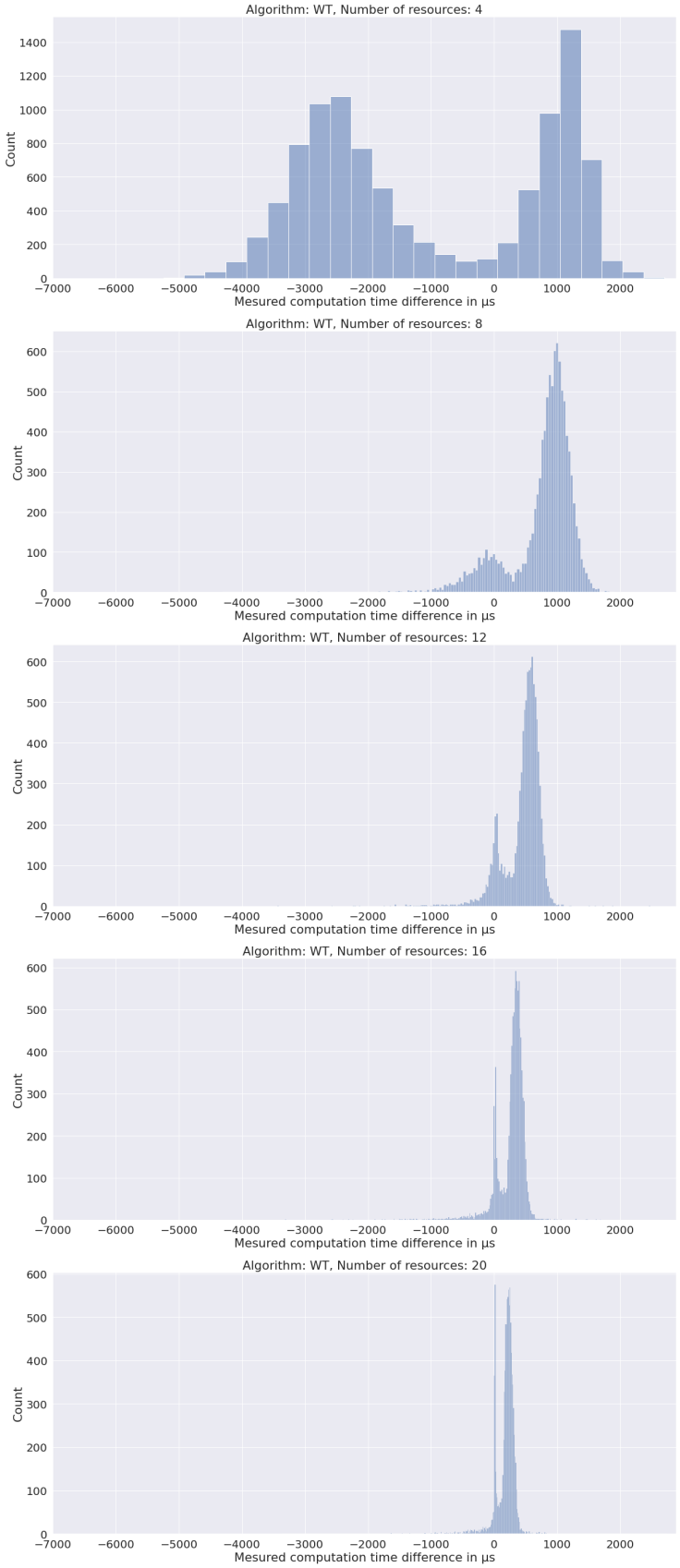



Figure 2.12: **Histograms presenting frame duration reductions in µs compared to FIFO for 4, 8, 12, 16 and 20 cores with multi-level scheduler for WT. (Positive values mean that the scheduler provides a shorter frame duration.)**

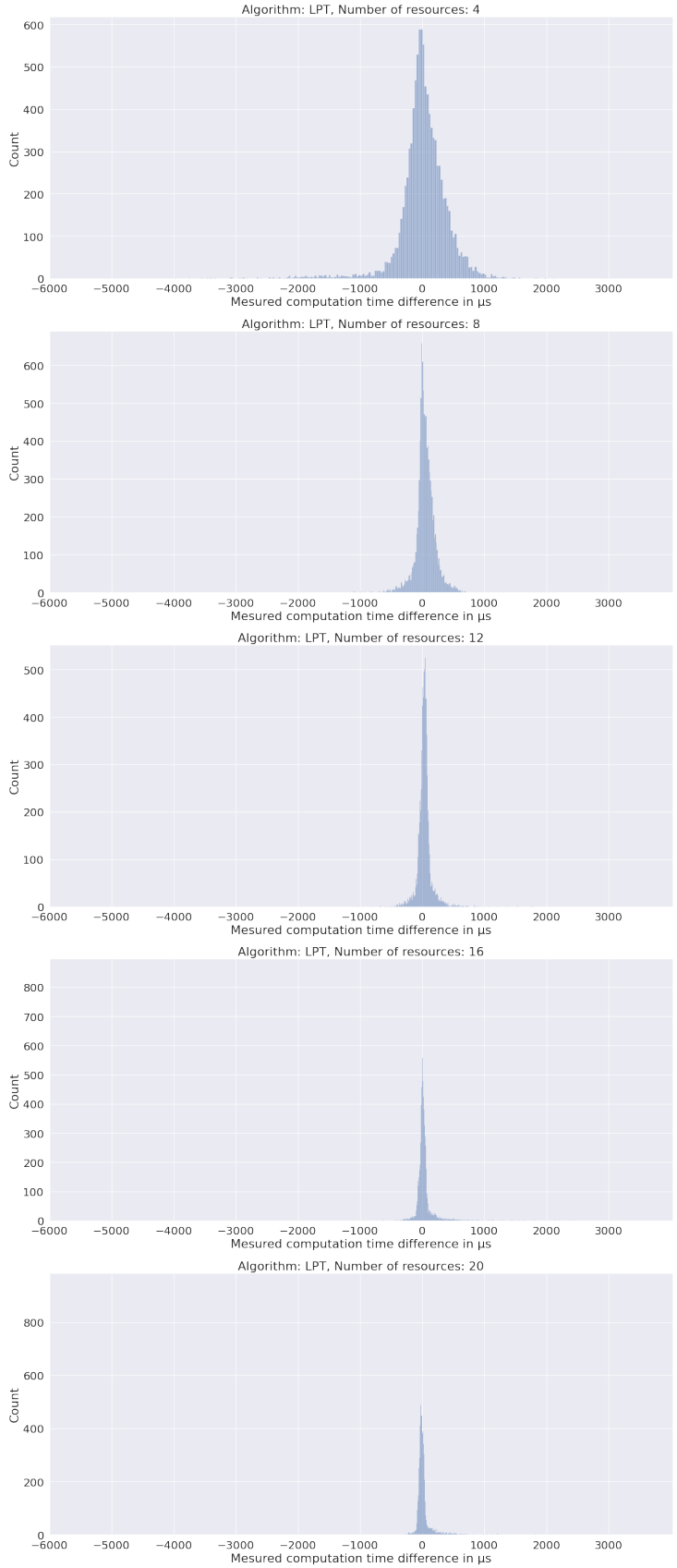



Figure 2.13: **Histograms presenting frame duration reductions in µs compared to FIFO for 4, 8, 12, 16 and 20 cores with multi-level scheduler for LPT. (Positive values mean that the scheduler provides a shorter frame duration.)**

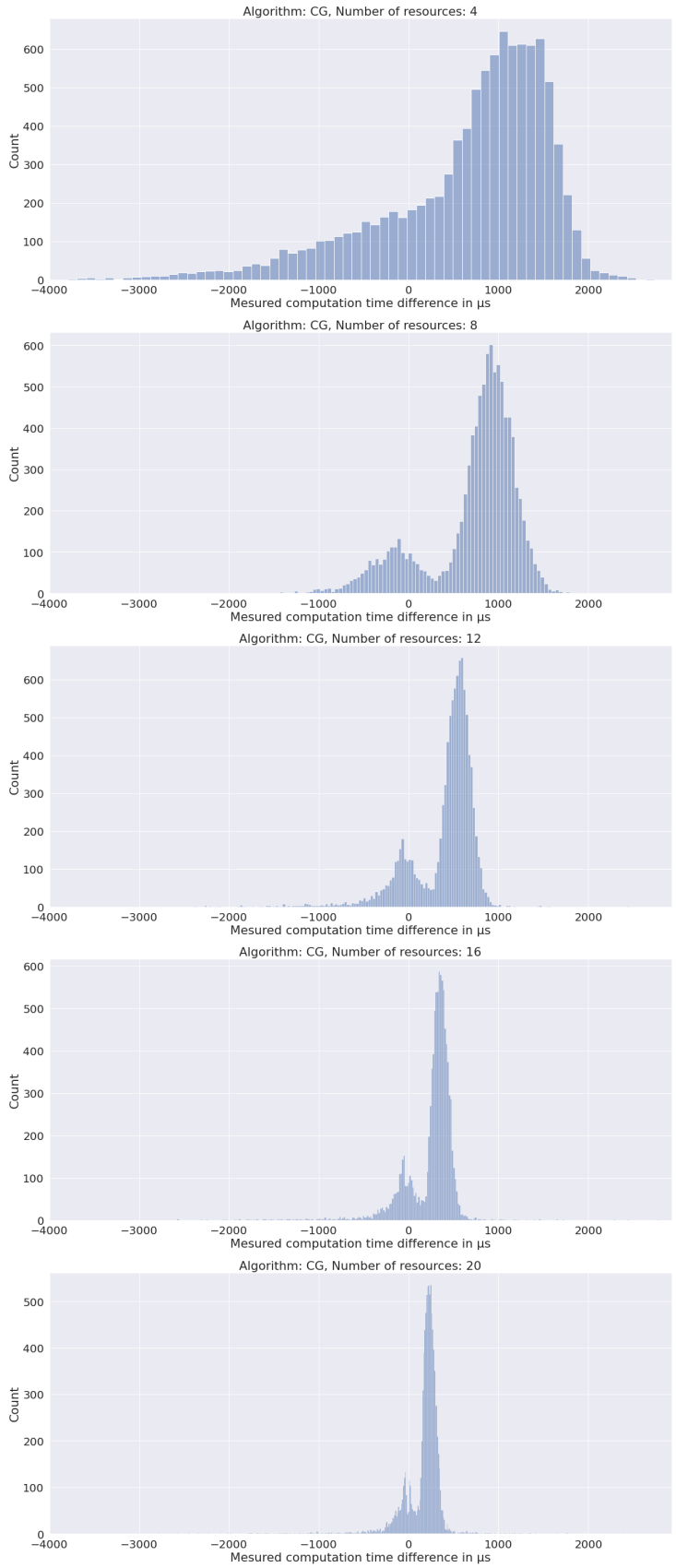



Figure 2.14: **Histograms presenting frame duration reductions in µs compared to FIFOfor 4, 8, 12, 16 and 20 cores with multi-level scheduler for CG. (Positive values mean that the scheduler provides a shorter frame duration.)**

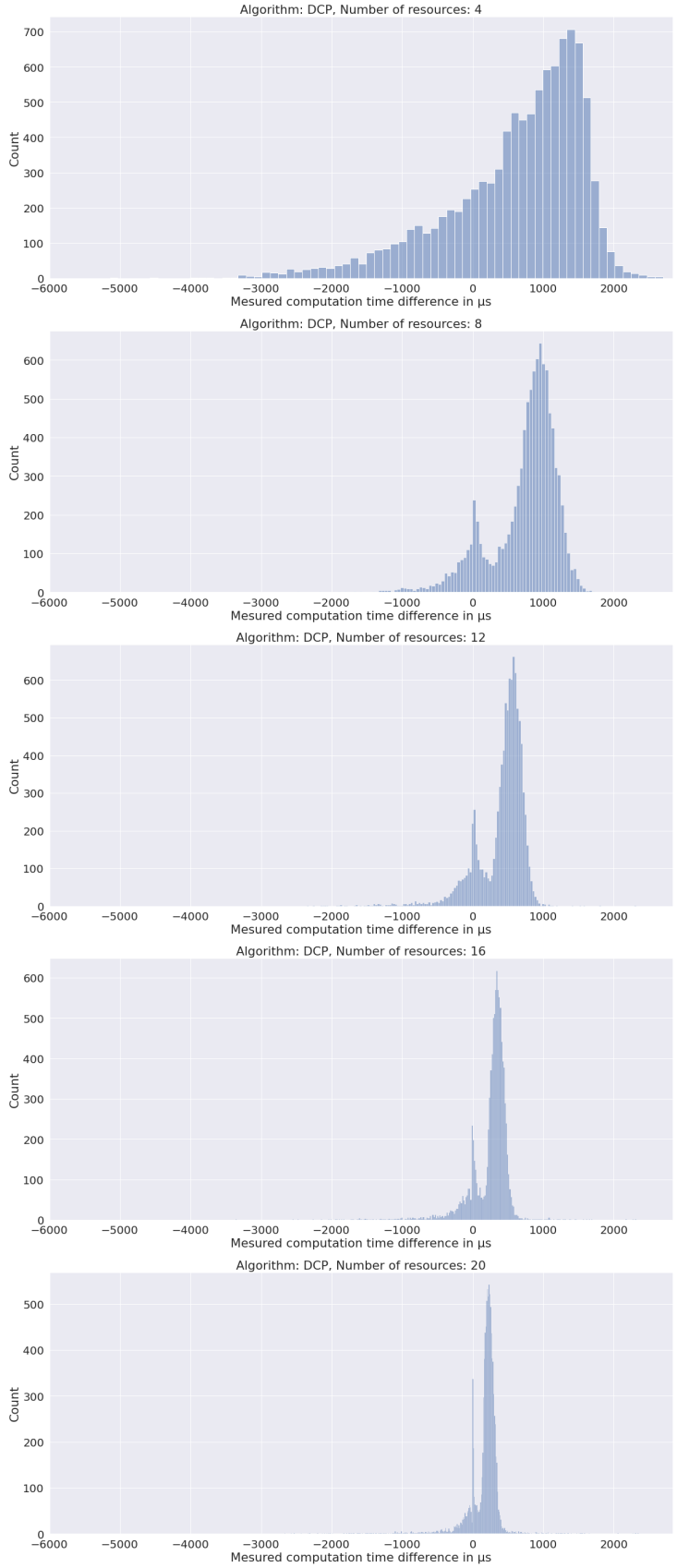



Figure 2.15: **Histograms presenting frame duration reductions in µs compared to FIFOfor 4, 8, 12, 16 and 20 cores with multi-level scheduler for DCP. (Positive values mean that the scheduler provides a shorter frame duration.)**

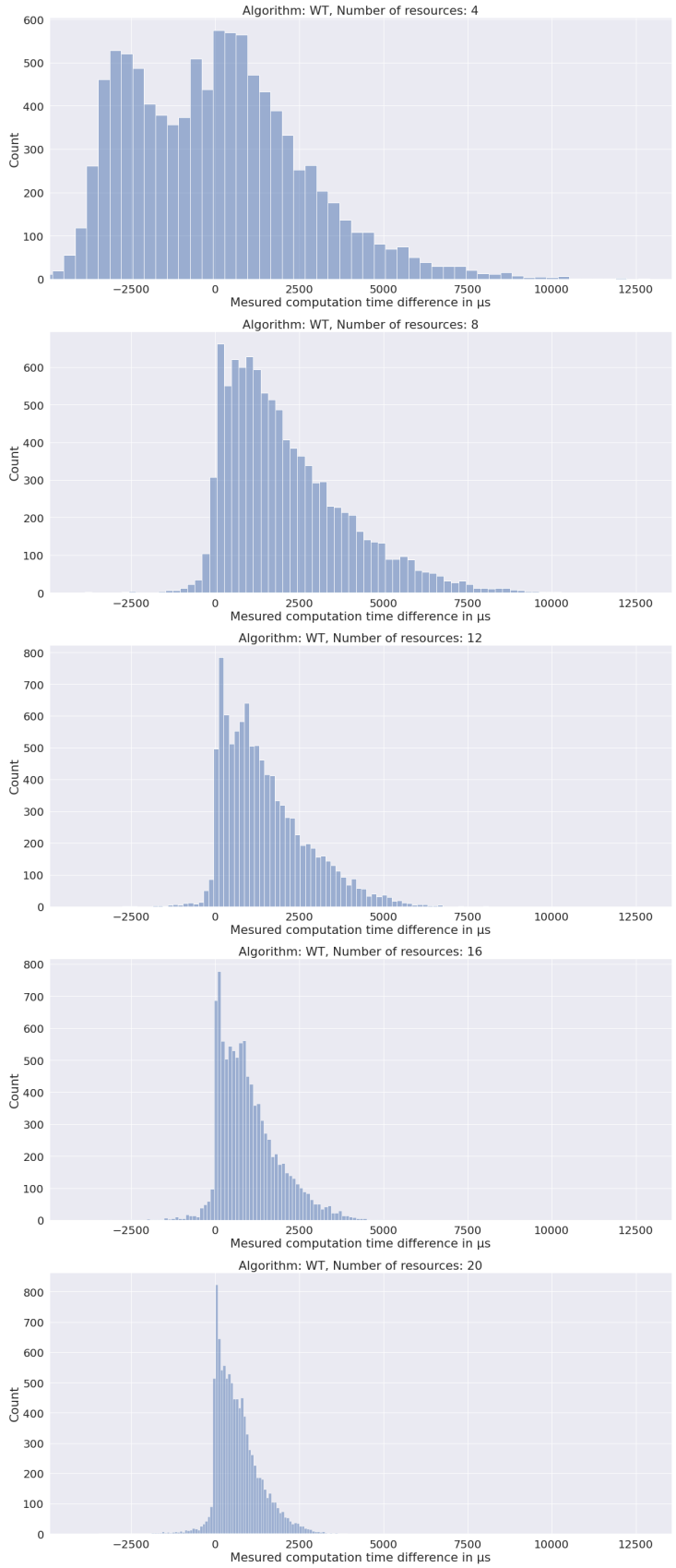



Figure 2.16: **Histograms presenting frame duration reductions in µs of scheduling with multi-level scheduler compared to baseline for 4, 8, 12, 16 and 20 cores for WT. (Positive values mean that the scheduler provides a shorter frame duration.)**

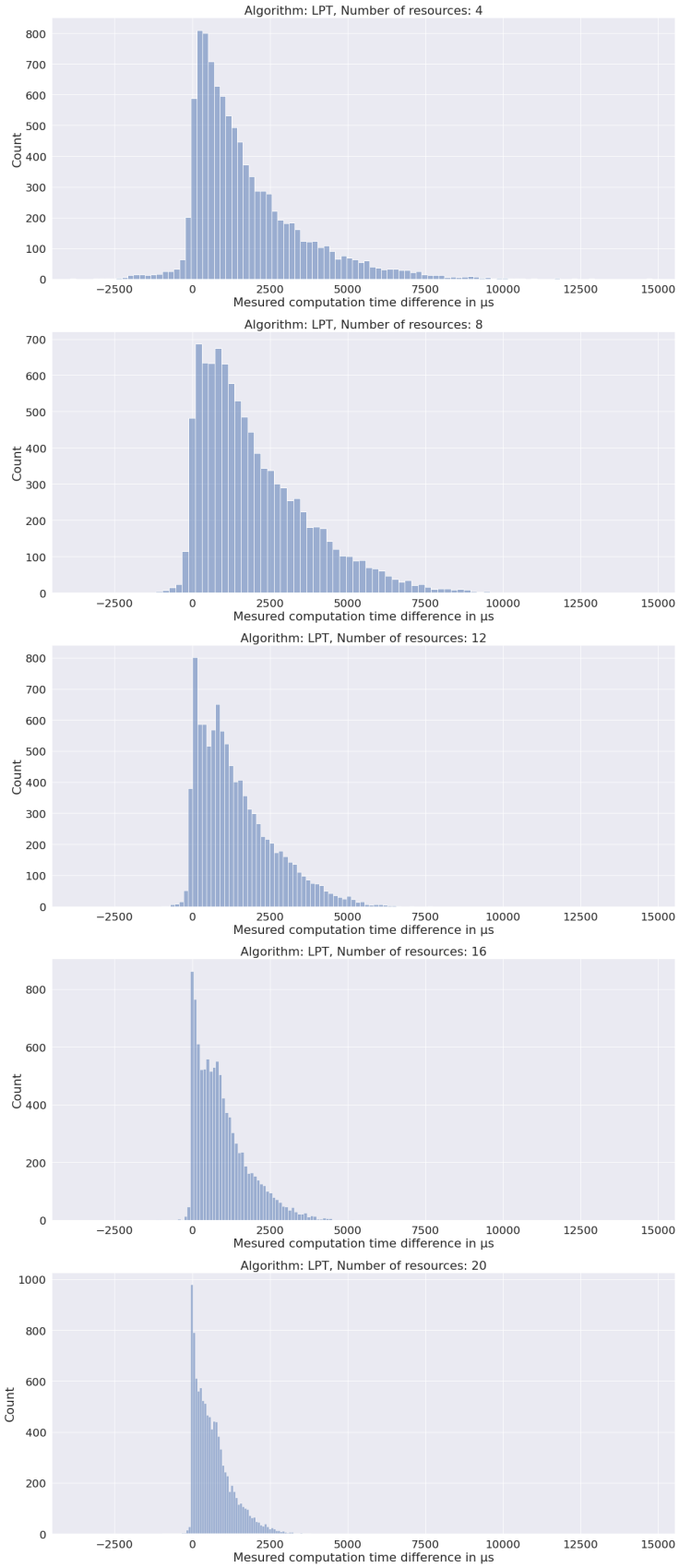



Figure 2.17: **Histograms presenting frame duration reductions in µs of scheduling with multi-level scheduler compared to baseline for 4, 8, 12, 16 and 20 cores for LPT. (Positive values mean that the scheduler provides a shorter frame duration.)**

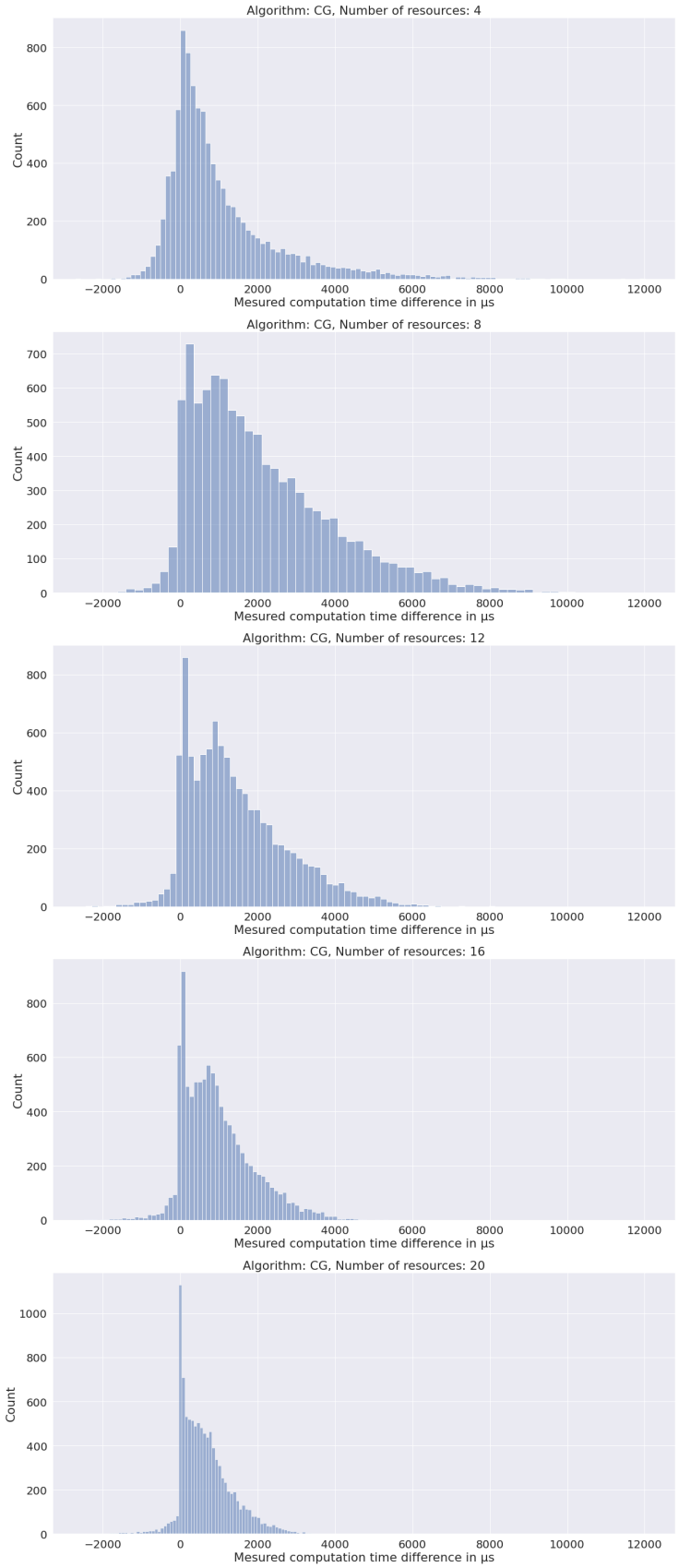



Figure 2.18: **Histograms presenting frame duration reductions in µs of scheduling with multi-level scheduler compared to baseline for 4, 8, 12, 16 and 20 cores for CG. (Positive values mean that the scheduler provides a shorter frame duration.)**

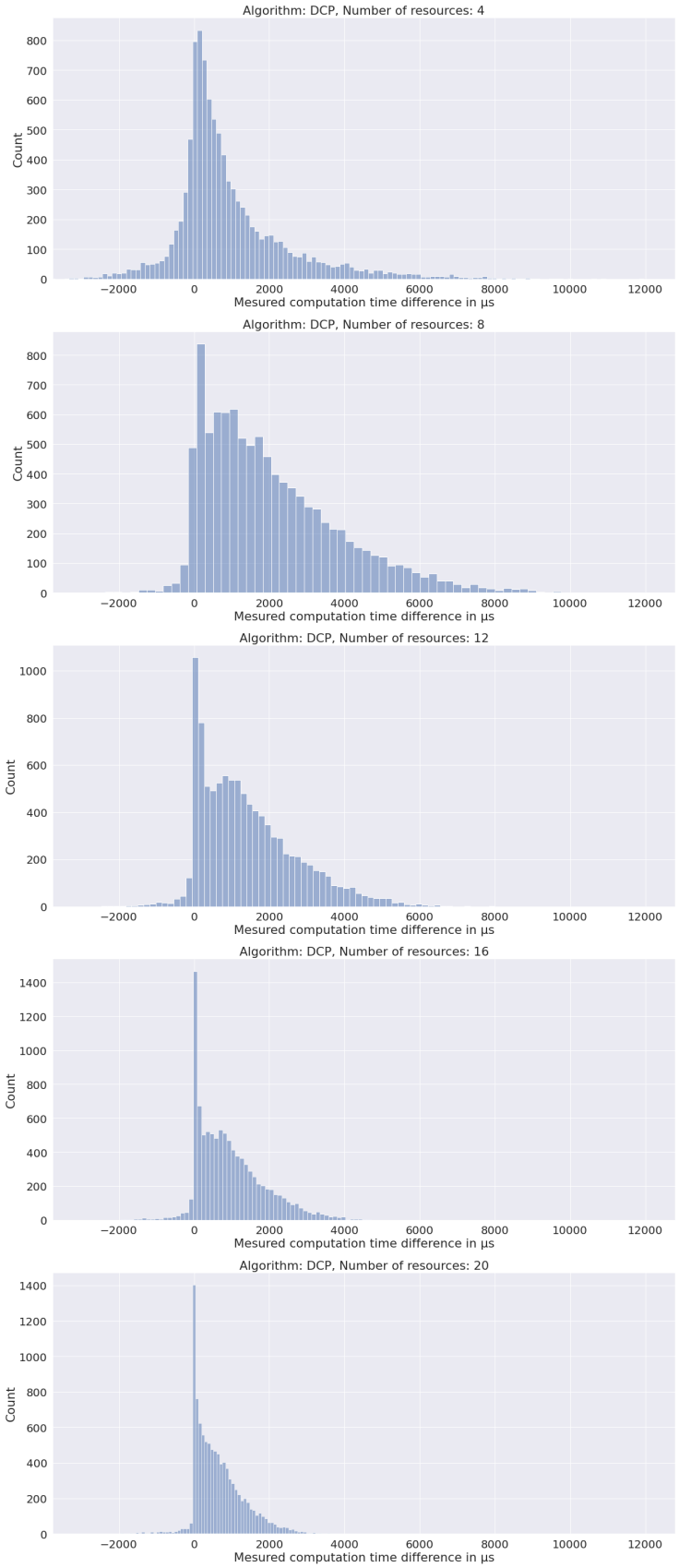



Figure 2.19: **Histograms presenting frame duration reductions in µs of scheduling with multi-level scheduler compared to baseline for 4, 8, 12, 16 and 20 cores for DCP. (Positive values mean that the scheduler provides a shorter frame duration.)**

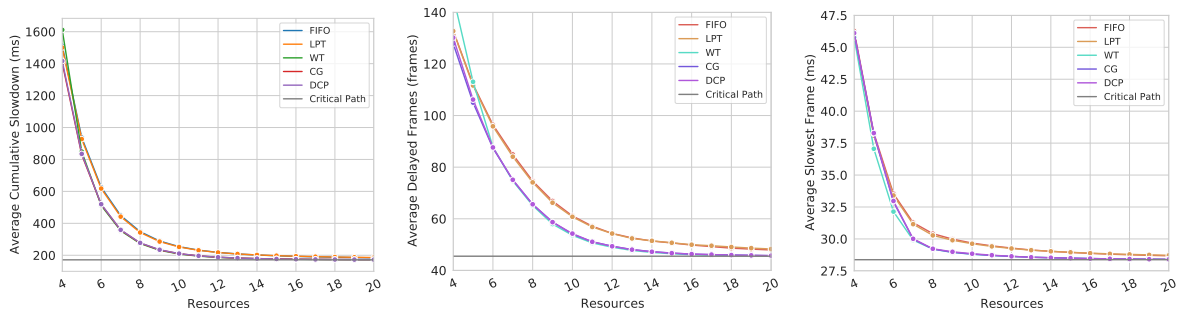



Figure 2.20: **Average number of cumulative slowdown, delayed frames and slowest frames duration for a subset of the schedulers on different number of resources. The vertical axis starts at 40 frames to emphasize differences. An overlap occurs in the plot, between FIFO and LPT (higher values) and WT CG DCP (lower values)**

loss in performance for a small amount of frames on lower amount of resources. This loss of performance shrink with the increase of the amount of resources. Moreover, in the same manner as in Scenario 1 they create a gap between their performance and the baseline that tends to decrease when many resources are available. Nonetheless, when compared to Scenario I, several major differences are present here. First, the absolute difference in values for all metrics between the best strategies and FIFO are not strictly decreasing anymore. This comes from FIFO (and schedulers with a similar performance) benefiting more from the sorted subtasks for small numbers of resources. For instance, if we compare the results in Figs. 2.20 DF and, 2.9, we can see that sorting subtasks reduces the number of delayed frames on 4 resources by about 5 for FIFO (from 138.26 to 132.9) but only 3 for CG (from 131 to 127.9). In these situations, the act of sorting reduces the impact of bad scheduling decisions, but this effect tended to disappear as the number of resources increased.

Then, comparing Scenario 1 and, 2 Fig.2.17 Fig.2.19 show a clear improvement on the scheduling. We can notice in nearly all plot gain in performance more significative than changing scheduling heuristics with little to no loss even if we can notice that WT algorithm under-performs for 4 resources. These gain follow the same pattern as previously benefiting more on lower amount of resources.

Finally, the last major difference comes from the virtual disappearance of the gap between the best schedulers and the Critical Path for all considered metrics. If we consider CG running over 16 resources, the average differences are 0.09, 0.82, and 3.97 for the Slowest Frame, Delayed Frames, and Cumulative Slowdown metrics, respectively. These proximity of the results to the optimal solution highlight the benefits of using scheduling algorithms and internal scheduling mechanisms that are well-adapted to the problem being faced. It does not, however, lead by itself to a situation where 60 fps can be achieved under the worst lag situations.

Finally we can notice than even if the load variation is greatly contained using both scheduling heuristics and subtasks sorting. The frame duration is still not guaranteed to respect the pre-determined frame limit of 16.6ms. However, we also notice that sorting subtasks issued from classical fork-join situation on execution, results in a near optimal

solution. In this context there are no scheduling solutions able to reduce furthermore the span of the frames as it nearly fits the critical path. The solution is then to increase directly the parallelism of the engine.

In the following section, we study the last scenario in which we slightly extend parallelism. This provides us evaluation on the impact of parallelization of targeted tasks of the engine on the overal execution.

## 2.7 Changing tasks - Increased Parallelism

As the use of scheduling algorithms and the changes to the game engine scheduler have led us to near-optimal performance, the only way forward to improve performance even more requires a new optimal. That, in its turn, demands changing the task graph. We have identified the two tasks with the longest processing times and chosen to simulate how an increase in their parallelism would affect performance. For each subtask in these tasks, we run two subtasks, each with half of their original processing time. We have chosen for this approach because it minimizes the changes to the overall task graph — namely, it does not change the dependencies nor the total processing time of the tasks, and it does not require modifying most of the tasks. Nevertheless, we are aware that these changes may not be feasible in some game engines due to the nature of the tasks being computed.

The performance results achieved in this scenario using 12 resources are summarized in Table 2.4. It follows the same organization of Table 2.3. The increase in parallelism in the two most critical tasks leads to performance improvements for all scheduling strategies and metrics. When compared to Scenario I, the strategies' Slowest Frame was decreased by about one quarter, their number of Delayed Frames was reduced by over one half, and their Cumulative Slowdown was reduced by about three quarters. When comparing FIFO's performance in Scenarios II and III, these metrics are improved by factors of 1.174, 1.602, and 2.390, respectively, which are proportionally larger than the improvements seen from Scenario I to II.

| | FIFO | LPT | SPT | SLPT | SSPT | HRRN | WT | HLF | HLFET | CG | DCP | Critical Path |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SF (ms) | 24.94 | 24.92 | 24.32 | 24.60 | 24.81 | 24.31 | 24.29 | 24.40 | 24.45 | 24.30 | 24.32 | 22.99 |
| *(% change)* | -24.13 | -24.18 | -24.92 | -24.02 | -24.31 | -24.90 | -25.01 | -24.86 | -24.87 | -24.97 | -24.92 | -18.97 |
| DF (frames) | 33.90 | 33.88 | 28.74 | 31.10 | 32.94 | 28.44 | 28.3 | 29.54 | 30.44 | 28.48 | 28.62 | 17.92 |
| *(% change)* | -53.23 | -53.50 | -58.24 | -54.73 | -54.26 | -58.48 | -58.83 | -57.51 | -56.50 | -58.57 | -58.34 | -60.62 |
| CS (ms) | 91.19 | 90.80 | 73.35 | 81.35 | 87.45 | 72.37 | 71.87 | 75.47 | 77.65 | 72.42 | 72.75 | 41.81 |
| *(% change)* | -75.70 | -75.90 | -78.70 | -76.29 | -76.42 | -78.90 | -79.02 | -78.41 | -78.05 | -78.90 | -78.78 | -75.60 |

Table 2.4: **Average metrics for all schedulers over 12 resources with sorted subtasks and additional parallelism in two tasks. Percentage reductions are calculated in comparison to Table 2.2.**

We provide the same set of plots as previous section in order to compare the performance of the heuristics in this scenario. **Fig 2.22** provides boxplots comparing the different metrics for various amount of resources, **Fig 2.24** and, **Fig 2.26** represent the comparison between FIFO and the different heuristics, providing hints on the performances of the heuristics in the context of increased parallelism, **Fig 2.28** and, **Fig 2.30** represents the comparison between similar heuristics for Multi-level scheduling described in previous section and the current optimization. These plots allows to distinguish gains in performance induced by the heuristics and, those caused by the changes in the task

graph's parallelism.

We notice that all metrics are strongly impacted by the increase in parallelism, **Fig 2.21** provides the evolution of the metrics depending on the amount of resources. The results showcase the ability to surpass prior constraints, as the average number of delayed frames decreases to less than 43 frames for 9 cores and more, the average slowest frame duration is reduced to under 27.5ms and, the average cumulative slowdown dips below 200ms with at least 8 cores available.

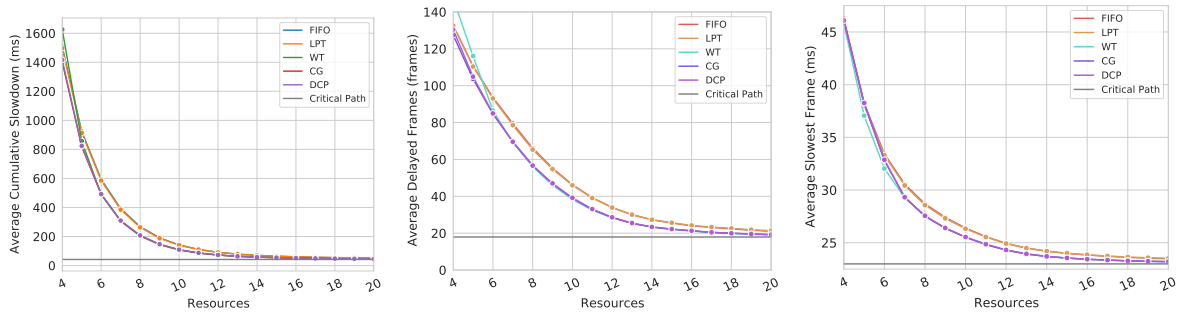


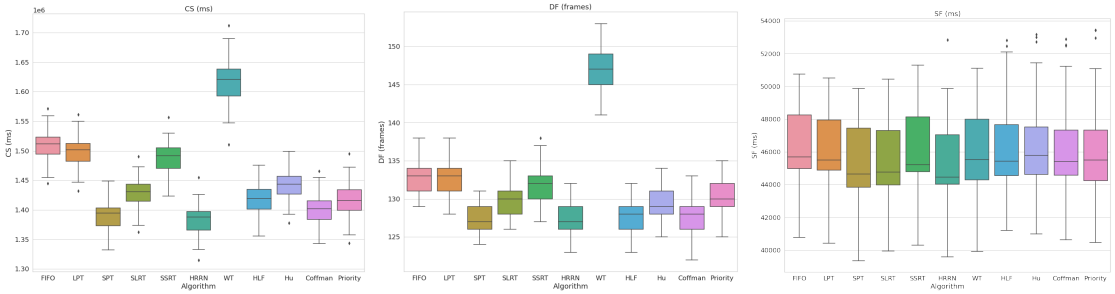

Figure 2.21: **Average number of cumulative slowdown, delayed frames and slowest frames duration for a subset of the schedulers on different number of resources. The vertical axis starts at 40 frames to emphasize differences.**

When we compare the scheduling strategies to their baseline (FIFO), their general behavior remains the same. However, we can notice a slight increase in the gains provided by the different heuristics for every amount of resources in comparison to gains with only multi-level scheduling. This highlight the capacity of heuristics to provide more performance with more parallelism. For example, strategies WT, CG, and DCP show better results than FIFO (all p-values $< 0.05$), while this cannot be said for LPT for metrics SF and DF (p-values $= 0.07$ and $0.87$, respectively). WT performed better than DCP for all metrics (all p-values $< 0.05$), but it performed the same as CG for metrics SF and DF (p-values $= 0.31$ and $0.06$, respectively). In the same manner we compare frame duration earlier, we now use CG as baseline algorithm instead of FIFO in **Fig 2.24** and, **Fig 2.26**. These figures highlight CG performances but also show that other algorithm perform better on specific frames providing more information on frame variation and changes in the critical path.

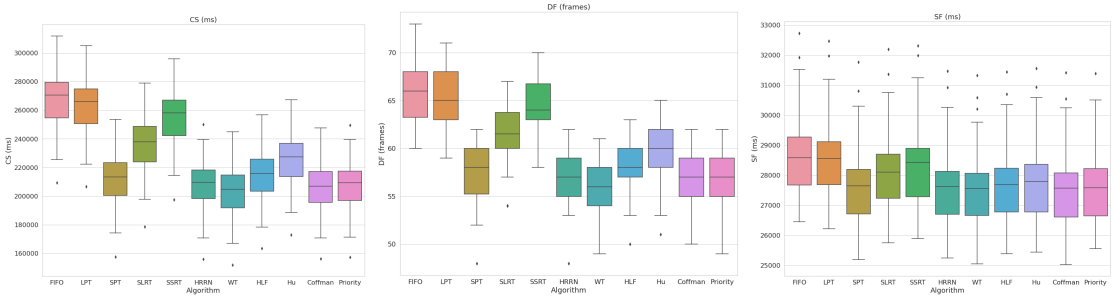
We also notice in **Fig 2.28** and, **Fig 2.30** a different behavior in the increase in performance. While multi level scheduling seemed to provide a substantial gain on lower amount of resources (between 4 and 12 resources) with diminishing gains as the amount of resources rises. The increase in parallelism provides lesser impact on lower amount of resources but increases with more cores. This highlight the complementary aspects of both these scenarios to enhance game engine performance.

We can notice that the gap between the strategies' performance and the new Critical Path has increased with the additional parallelism. In general, the best performing algorithms running over 12 resources still have average differences of approximately 1.5, 10.5, and 30.5 for the SF, DF, and CS metrics, respectively. This is mainly caused by a lack of resources, as 12 of them are not enough to profit from the additional parallelism in the tasks. When we increase the number of resources to 20, these differences are reduced to 0.2, 1.3, and 4.0, respectively.

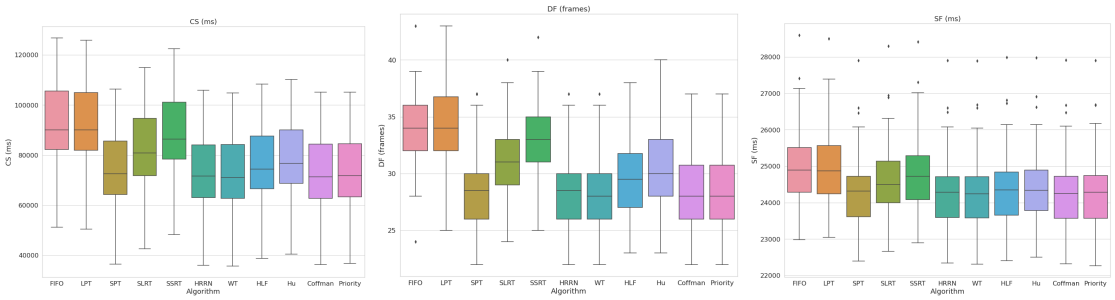
This evolution can be visualized in **Fig. 2.31**, which shows the change in average



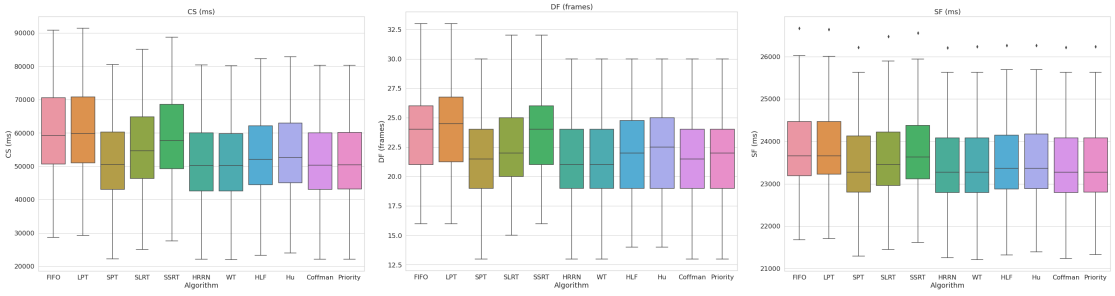

(a) 4 cores



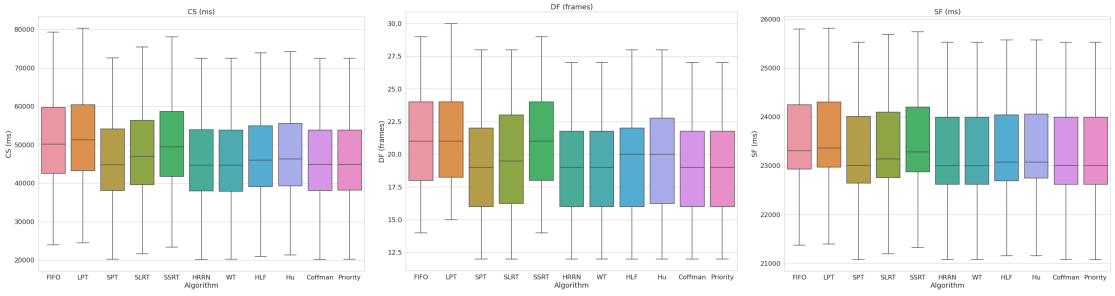

(b) 8 cores




(c) 12 cores




(d) 16 cores




(e) 20 cores

Figure 2.22: **Boxplots of the different metrics on 4, 8, 12, 16, 20 resources with multi-level scheduler and increased parallelism. (Vertical axes start at different points to emphasize differences.)**

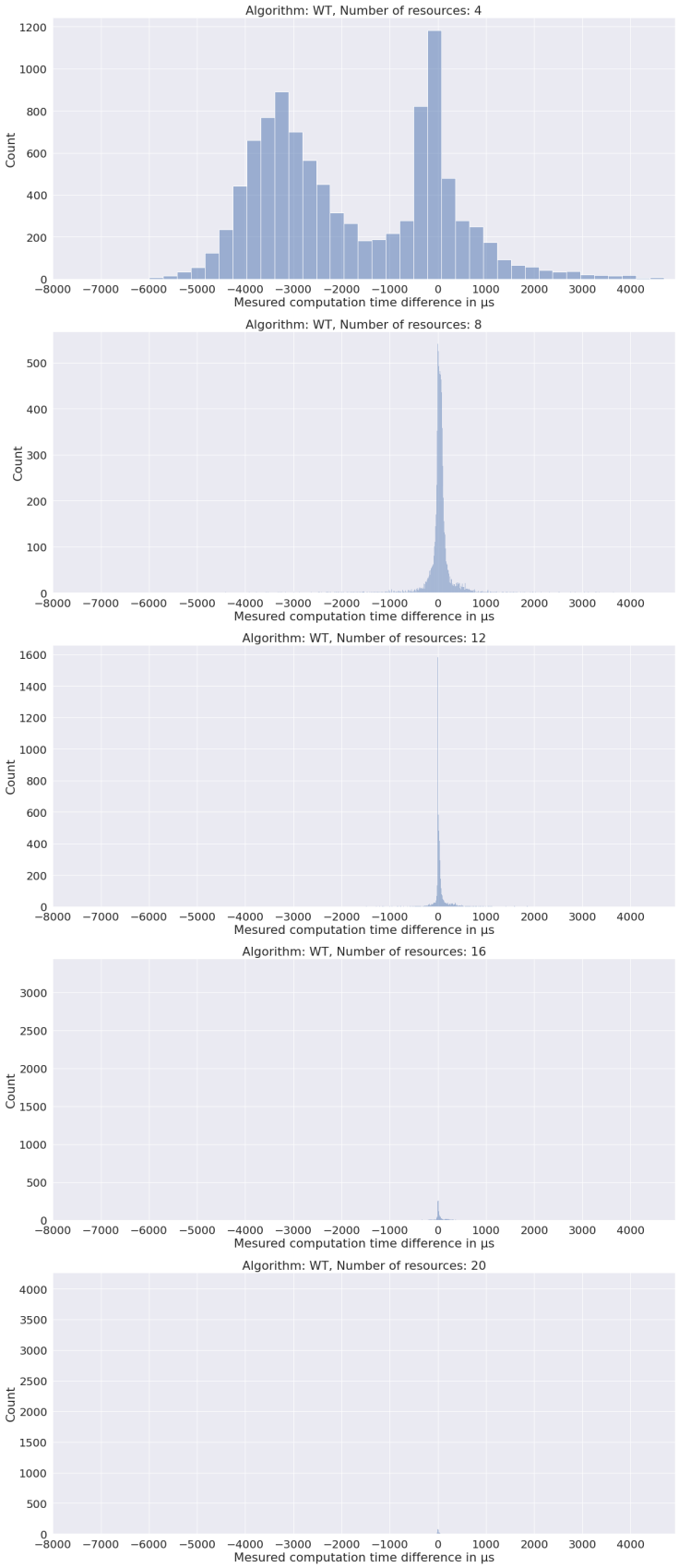



Figure 2.23: **Histograms presenting frame duration reductions in µs compared to CG for 4, 8, 12, 16 and 20 cores with multi-level scheduler and increased parallelism for WT. (Positive values mean that the scheduler provides a shorter frame duration.)**

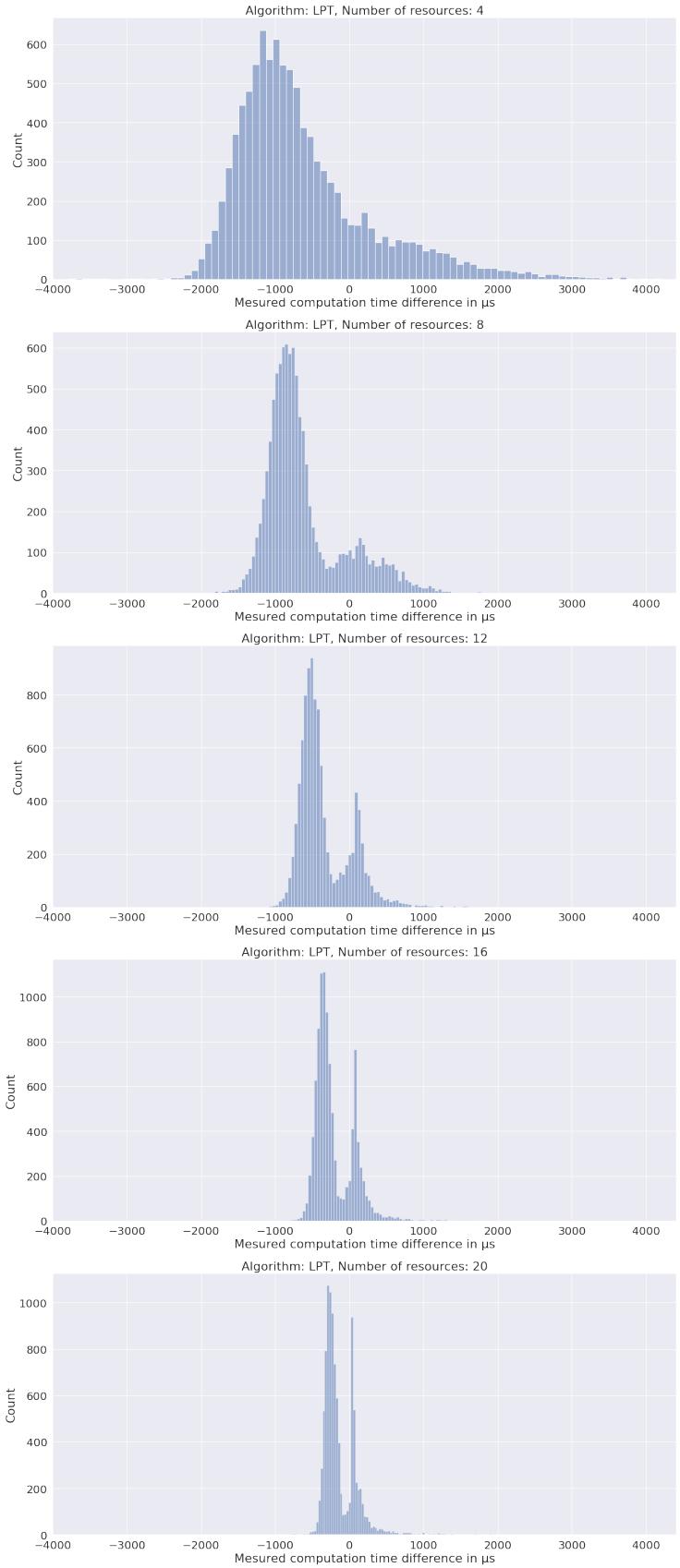



Figure 2.24: **Histograms presenting frame duration reductions in µs compared to CG for 4, 8, 12, 16 and 20 cores with multi-level scheduler and increased parallelism for LPT. (Positive values mean that the scheduler provides a shorter frame duration.)**

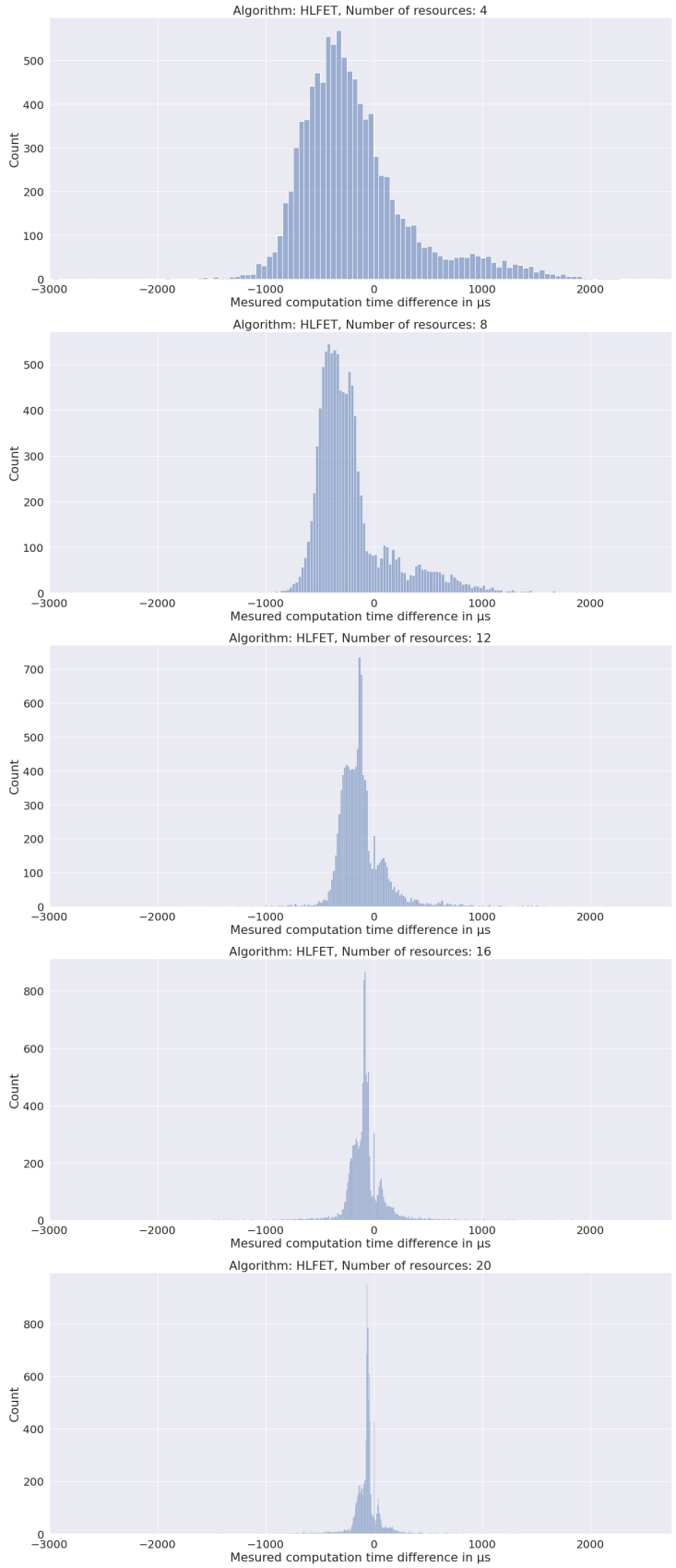



Figure 2.25: **Histograms presenting frame duration reductions in µs compared to CG for 4, 8, 12, 16 and 20 cores with multi-level scheduler and increased parallelism for HLEFT. (Positive values mean that the scheduler provides a shorter frame duration.)**

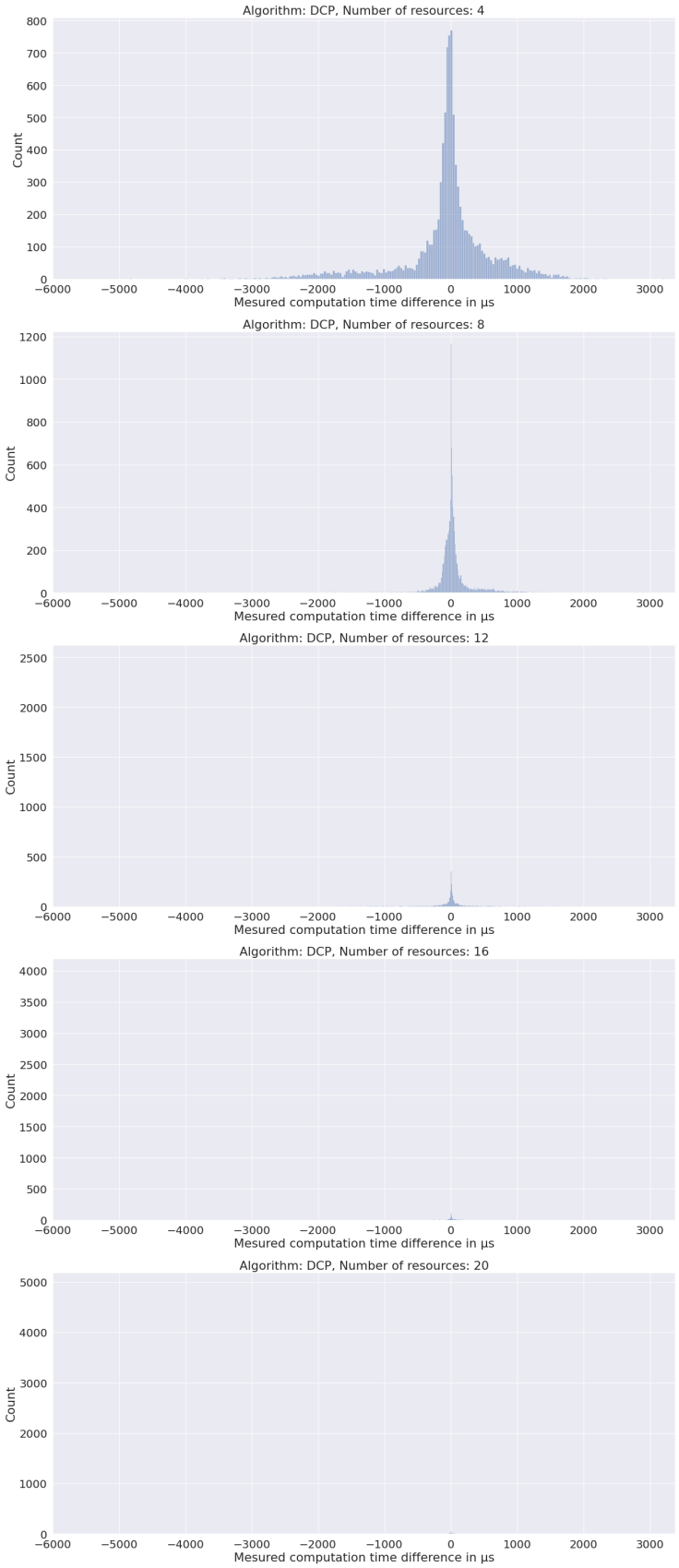



Figure 2.26: **Histograms presenting frame duration reductions in µs compared to CG for 4, 8, 12, 16 and 20 cores with multi-level scheduler and increased parallelism for DCP. (Positive values mean that the scheduler provides a shorter frame duration.)**

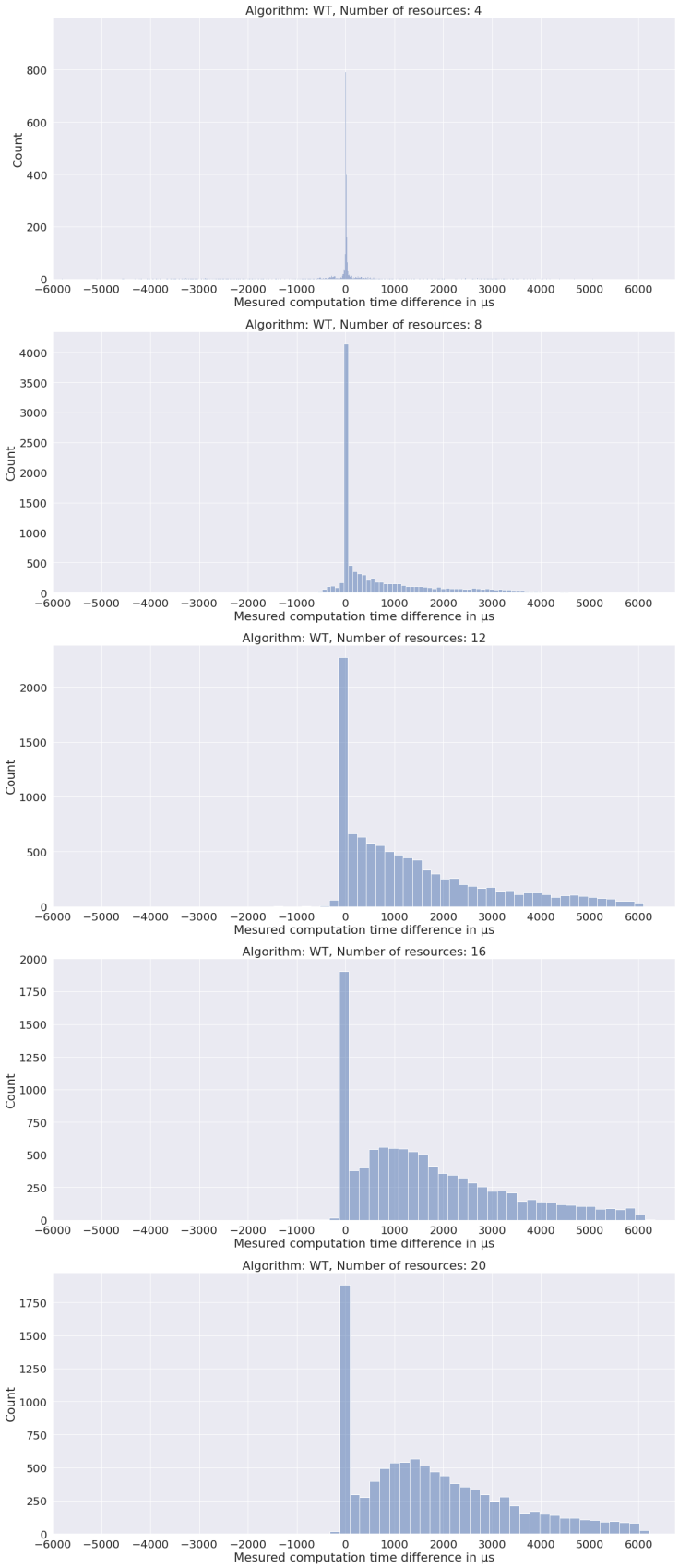



Figure 2.27: **Histograms presenting frame duration reductions in µs of scheduling with increased parallelism compared to multi-level scheduling for 4, 8, 12, 16 and 20 cores for WT (Positive values mean that the scheduler provides a shorter frame duration.)**

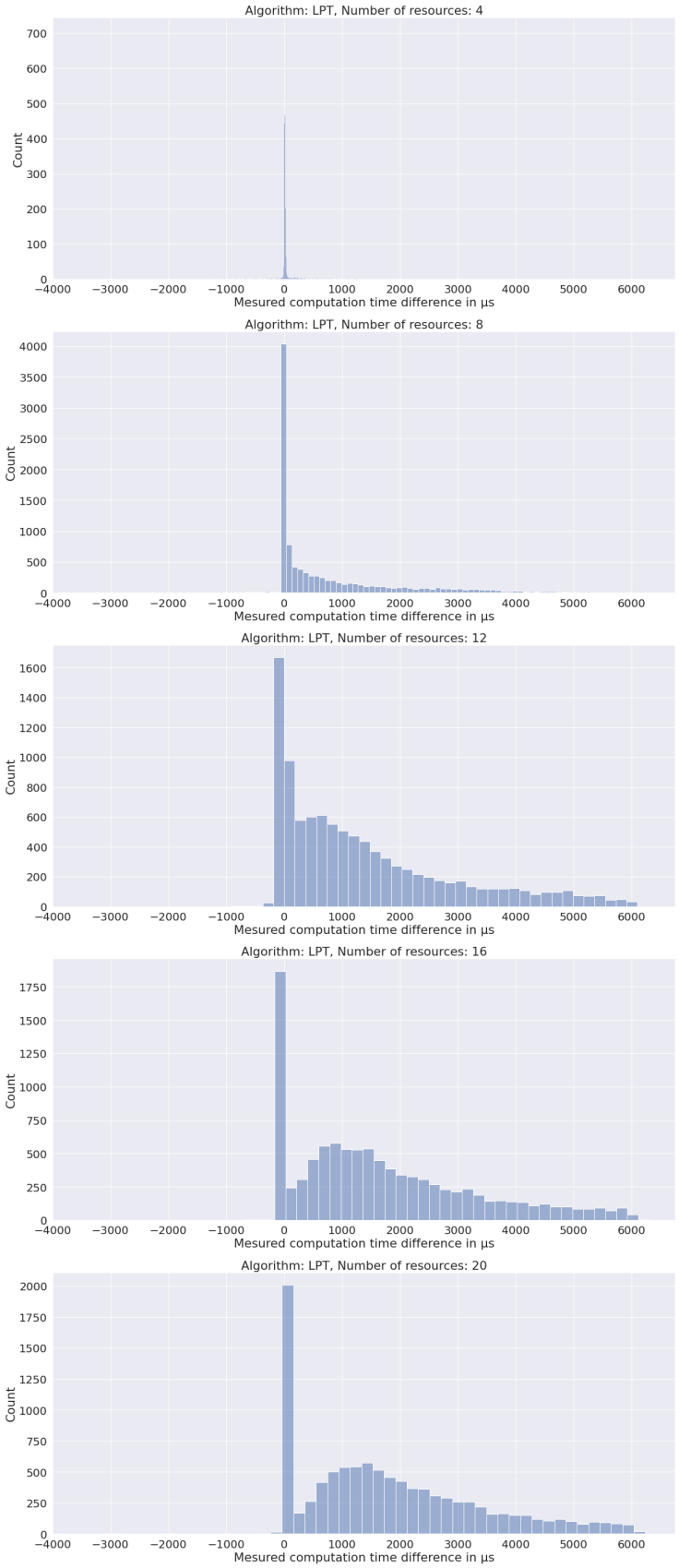



Figure 2.28: **Histograms presenting frame duration reductions in µs of scheduling with increased parallelism compared to multi-level scheduling for 4, 8, 12, 16 and 20 cores for LPT (Positive values mean that the scheduler provides a shorter frame duration.)**

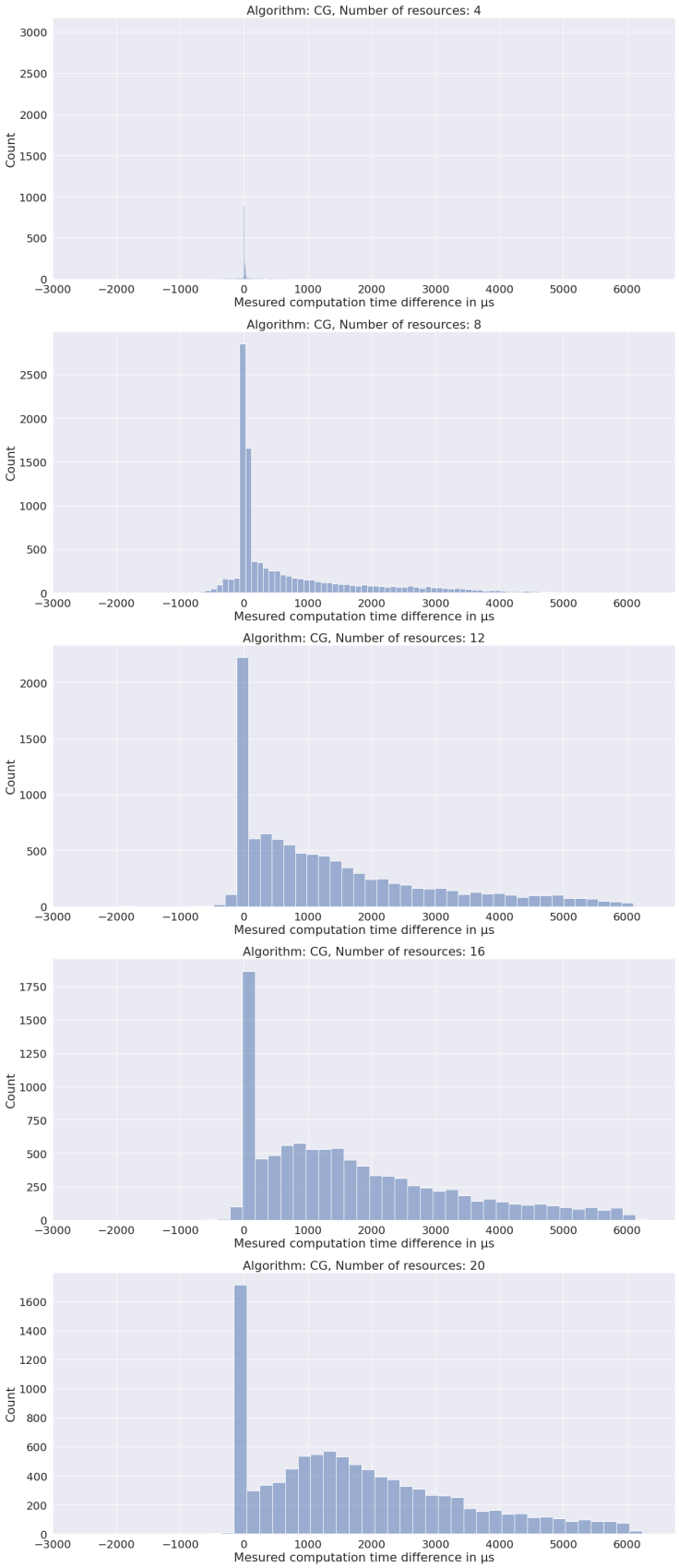



Figure 2.29: **Histograms presenting frame duration reductions in µs of scheduling with increased parallelism compared to multi-level scheduling for 4, 8, 12, 16 and 20 cores for CG (Positive values mean that the scheduler provides a shorter frame duration.)**

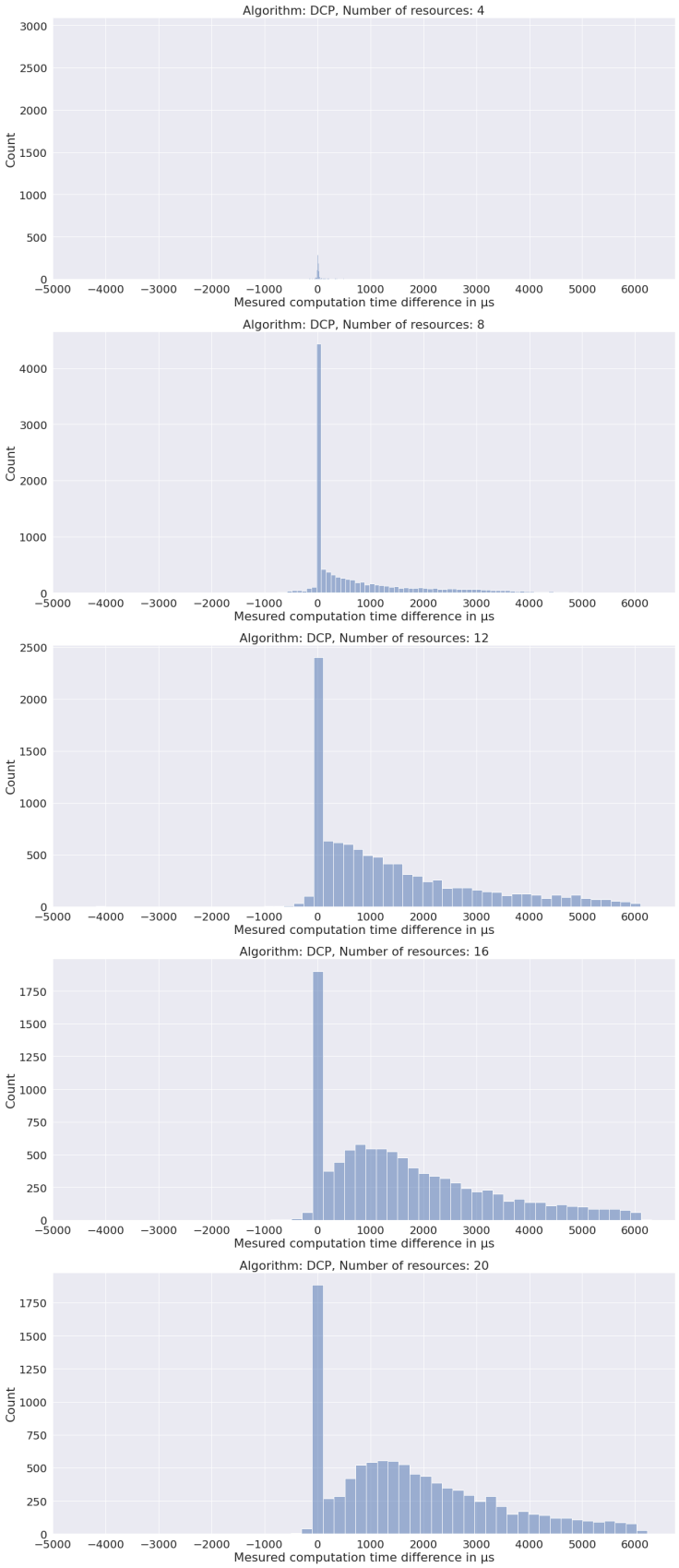



Figure 2.30: **Histograms presenting frame duration reductions in µs of scheduling with increased parallelism compared to multi-level scheduling for 4, 8, 12, 16 and 20 cores for DCP (Positive values mean that the scheduler provides a shorter frame duration.)**

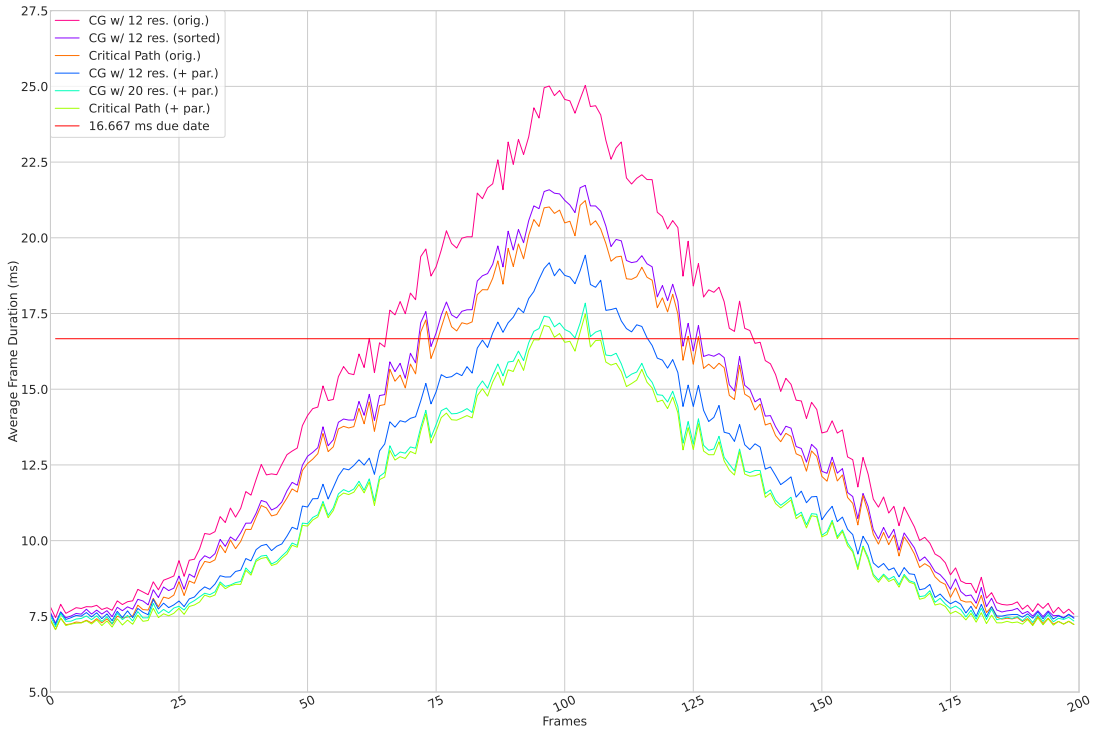



Figure 2.31: **Average Frame Duration of each frame for CG on 12 and 20 resources with baseline, multi-level scheduling and extra parallelism and the Critical Path.**

frame duration throughout the simulations for the three scenarios for 12 and 20 resources. (it is organized in a similar fashion to Fig. 2.10). As the figure illustrates, sorting the subtasks when scheduling, provides a first substantial gain between CG and the Critical Path. However this reduction in frame span does not enable to respect the frame limit of 16.667ms. The increase in parallelism introduced in this section provides a new gap in performance, allowing to overcome the previous critical path even with 12 cores even though it part of the execution still remains above the frame's drop threshold. However, this limitations is overcome by providing the engine more resources (20 cores).

Overall, we can clearly see that the improvements brought in each scenario makes the game engine more robust to computation's overload, leading to a better experience to users. Through all these different scenarios, we showed the differences between classical heuristics applied to specific graph issued from real games in order to diminish frame span and avoid frame drop when the engine is overloaded. We also proposed and evaluated the impact of sorting subtasks issued from classical fork-join situation on execution, resulting in a near optimal solution. In this context we considered the two major time-computing tasks as a set of parallel one, providing more parallelism to the engine, in this case, a noticeable gap between execution and critical path appears, requiring more resources to be fully exploited.

## 2.8 Conclusion

In the current state of video game engine, we provided experiments showing that several static heuristics show nearly optimal results, making it irrelevant to use dynamic plani-

fication for now. However we also noticed that even though several heuristics provided enhanced performance in average, FIFO heuristic could outperform them in some cases. This leads us to believe that in some cases changes occuring in the critical path of the engine may be taken into account to improve heurisitcs and bring maximum performance.

We studied extensively variations of computations for a given type of overload caused by AI. Nevertheless, there are different type of overload that may occur during the execution. These overload can be caused by various reactions of the engine to player actions. For instance we can cite, explosions or intensive collisions as shown in **Fig.2.32**. All these different type of overload will have similar effects as AI overload but they can also happen simultaneously.


Figure 2.32: **Example of intensive collisions in the game Binding of isaac. Every green "tears" are the main character's (in black on the screen) bullets. The game needs to compute the interactions between the Boss(in red) and the weapons every frames to compute damages. The massive number of bullets may cause the game to crash.**

In this context, we can notice that there are strong variation in the critical path in the execution depending on the phases of the game. One of the objective of the scheduler is to adapt to these different phase.

Nonetheless, we also have to take into consideration the different evolutions video games could face in the next years. For instance, the growth in the computation's capacity of devices running video games. Moore's law is an observation made by Gordon Moore, one of the founders of Intel, in 1965. It states that the number of transistors on a microprocessor chip doubles approximately every two years, while the cost of manufacturing the chip remains constant or decreases.[36] In the context of video games, Moore's law has had a significant impact on the performance and capabilities of microprocessors used in gaming hardware. As the number of transistors on microprocessors has increased over time, the processing power of gaming hardware has also increased. In this context, we can expect for video games consoles and personal computers processing capacity to keep increasing for coming years. This rise in computations capacity provides the possibility for game developers to create more complex worlds and games. These evolutions on both hardware and software areas forces a better repartition of computation among resources and a refined definition of tasks. Given the rapid growth of the sector and, the

constant evolution of game engines, we can consider that next generation of engine will require dynamic solution for scheduling. To be pertinent, these solutions need to take into account specifities and constraints of video game engines.

We also noticed some structural lack of parallelism on the beginning and end of the frame when the synchronization of buffers occurs. This provides a segment where computations are possibles but this segment is limited and we need to have a strong control over the time allowed to the scheduler. In the same manner, memory is one of the strong limitation in video game, in order to provide efficient and respecting the different constraint of the scheduler, we provide a method allowing time bounded and memory bounded dynamic scheduling.

# Chapter 3

# Dynamic adaptative scheduling for video games using Monte Carlo Graph Search

In the previous chapter, we highlighted the efficiency of several scheduling heuristics on the problem of video game. Some of these methods were static such as Coffman-Graham while providing satisfying performance. However, because of the evolution expected for next generations of engines and the direct impacts computation's variation have on the critical path a static scheduling will not be able to provide satisfying performance on the long term. This highlights the necessity to develop a dynamic scheduling method providing a better adaptation regarding the load of the game and resources available.

Due to video games limitations, the induced overhead as well as the amount of memory required by the scheduling method has to be strictly controlled. Nonetheless, some specifications of video games provide some advantages regarding the scheduling method. Even though the load variation in the game has a great impact on the frame duration and critical path, these variations occurs through slow transition phases. This allows for an iterative adaptation of the scheduling. Moreover, the relative continuity of video-games induces little to no variations in tasks duration from one frame to another. This leads to a relative stability of the engine providing a natural oracle from frame to frame.

To schedule such system, one idea is to explore, given a static heuristic known to perform relatively well, slight modifications of the scheduling iteratively for a fixed period of time. This exploration introduces a feed back loop, and using characteristics of the engine and previous results, improve the resulting scheduling frame per frame.

## 3.1 Monte Carlo Tree Search

In order to create such dynamic heuristic, we opted for a Monte Carlo Tree search algorithm as it fits all our constraints as well as having proven itself performing for highly computational problems.

### 3.1.1 Background on Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a family of simulation-based methods to select near optimal actions in a large exploration space. It combines exploration and exploitation using UCT(Upper Confidence bounds applied to Trees) in Kocsis an Szepesvári [26] as its most widely used variant. MCTS has been applied successfully for a number of sequential decision problems ([47] for a survey).

MCTS algorithm consists in building the decision tree of a problem using random simulations to guide future search decisions. It can be described by 4 sequential steps illustrated in **Fig.3.1**.

**Selection** : Selects the node that is to be explored. This selection aims to balance exploration (investigating unknown or less visited parts of the tree) and exploitation (focusing on areas of the tree that have yielded promising results so far).

**Expansion** : Creates nodes resulting from the different actions available among the selected node.

**Evaluation (also called Simulation)** : Complete path until it reaches a leaf of the decision tree (end of the problem) using random decisions. It then evaluates the resulting solution to determine a score (in a game such as Chest, for instance it could be win or lose with 1 or 0).

**Backpropagation** : The score resulting from the simulation is propagated in all visited nodes of the path. This provides more information for future cycle of the algorithm.

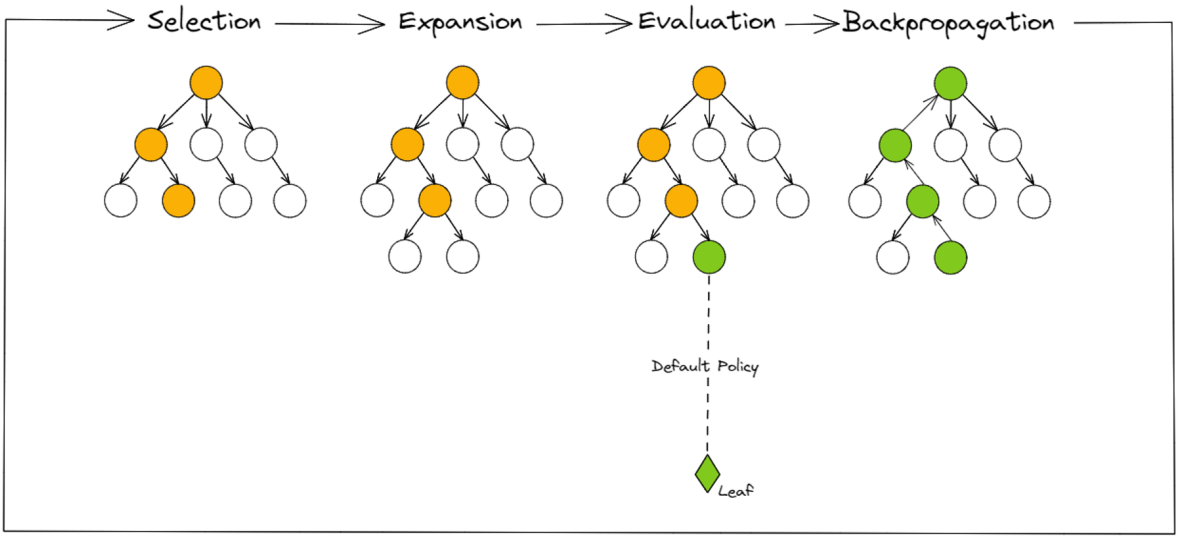



Figure 3.1: **Construction of the Monte Carlo Tree Search and Exploration mechanism**

One of the strong benefits of this method is that MCTS is an anytime algorithm, meaning that it can be stopped at any point during the search process and still provide a reasonably good solution. The quality of the solution improves as more time is allocated to the search, allowing the algorithm to adapt to varying computational budgets or time

constraints. Moreover MCTS refines the quality (score) of nodes iteratively through the backpropagation step. As the algorithm proceeds and more simulations are conducted, the quality of nodes become more accurate, allowing for better decision-making and search guidance.

In 2015, this method was paired with a Deep Neural Network to provide unreached performance in the Game of Go [37]. The deep neural network here simply replaced the "Simulation" phase by providing a probability to win that was used as scoring method to guide exploration.

### 3.1.2 Adapt Monte Carlo Tree Search to scheduling

So far, few works propose MCTS as a way to solve scheduling problems [6, 43] and none correspond to the scheduling problem we have.

The principle of the method we present in the following sections is to build a tree exploring different possible task orderings. The method is not designed to build a schedule, but only generate a priority order between tasks (one priority order per path in the tree). The schedule is then dynamic, following this order and allocating resources for tasks when they are available or waiting for some resources otherwise. The nodes of the tree correspond to the ready tasks that can be executed (the root contains only $t_1$) and edges fire a task, as in an automaton. Following a single path from root to leaf therefore provides a scheduling order (priorities). Sub-tasks composing a task are considered to have the same priority as their task. The space of all possible schedules, even for small graphs, is too large to be explored completely. This is the reason why we use a Monte Carlo Tree Search method, limiting the exploration only to the most promising branches. Equation 3.1 and 3.2 describes the amount of nodes created using a MCTS method for a Diamond structure context of k tasks illustrated in Fig.2.5.

$$V_0 = N \tag{3.1}$$

$$V_k = V_{k-1}.(N - k) \tag{3.2}$$

In the usual MCTS method, there is an exploration of the first levels of the tree explicitly and the remaining nodes of the paths are not created but handled by some fast method (Deep Neural Networks in the case of AlphaZero for instance). Here, for scheduling purposes, we propose to complete any partial path of the tree by the ordering defined by Coffman-Graham(CG) method. As CG is static and needs to be computed only once, the cost of completing a partial ordering is reduced. When starting the exploration the proposed scheduling order of this method is therefore the one proposed by CG. We create at the initialization a unique path in the tree corresponding to CG ordering. Then, after a few explorations, new schedules will appear, starting with a different task ordering and then finishing with the one in CG. By making an evaluation at each step of the most promising paths including among them the one for CG, this ensures that only schedules outperforming CG will be kept.

Finding a good ordering is therefore performed online, and continuously. This exploration can be controlled, the number of explorations being limited by the idle time available in the frame computation that is used by this exploration. We will show in

the evaluation that the method, while scheduling online and continuously, does not incur overheads and performs the exploration seamlessly.

## 3.2 Monte Carlo graph search

One of the difficulty of the schedule exploration with an MCTS is the number of possible solutions resulting from tasks permutations in the ordering. To tackle this complexity issue, Monte Carlo Graph Search (MCGS) proposed in Leurent and Maillard [29] and applied for instance to AlphaZero in Czech et al [17] reduces the space complexity by folding the tree into a graph, sharing the same nodes in different branches. This has not been applied to scheduling. In comparison to equations 3.1 and 3.2, equations 3.4 and, 3.4 shows MCGS method greatly reduces the amount of nodes in comparison to the MCTS.

$$V_0 = N \tag{3.3}$$

$$V_k = V_{k-1}.\frac{(N-k)}{k+1} \tag{3.4}$$

The idea of using an MCGS instead of the classic MCTS comes from a specificity of the scheduling problem. When exploring the MCTS solution for scheduling, we noticed the introduction of a great amount of symmetric branches as shown in **Fig.3.2**. These symmetries negatively impact performance by leading to redundant exploration of already known scheduling and consuming valuable memory resources due to the creation of superfluous nodes. This specificity arises directly from the symmetrical nature of priority ordering among independent tasks. Moreover due to strong limitations in memory in video-games and in order to make the algorithm viable in a production environment, reducing the complexity in space was a necessity.

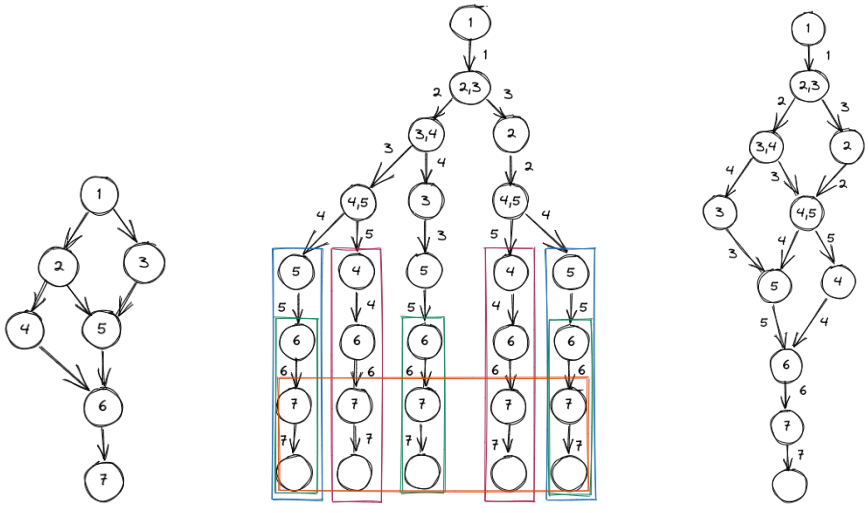



Figure 3.2: **(a). Example of task graph and (b). its corresponding representation of Monte Carlo Tree Search fully expanded with scheduling symetries represented by colored frames and (c). Corresponding Monte Carlo Graph Search**

### 3.2.1 MCGS Space and Time Overhead

To compare the amount of node created when using MCTS and MCGS and evaluate the gain in space, **Fig.3.3** represents the amount of exploration per node created while running on a video game task graph. By construction the MCTS algorithm constructs one new node for every exploration, while as seen on the plot, MCGS has two distinct phases. The first one, below 1000 explorations, follows the ratio $1:1$ between explorations and node created. This phase can be seen as the construction of the "core-graph" corresponding to the CG heuristic and some small variations around. The second phase, beyond 1000 explorations, follows a $\log(x)$ node creation for $x$ exploration evolution, this describe a more sporadic node creation. The amount of node created is reduced because they refer to nodes that have already been added while exploring previous paths.

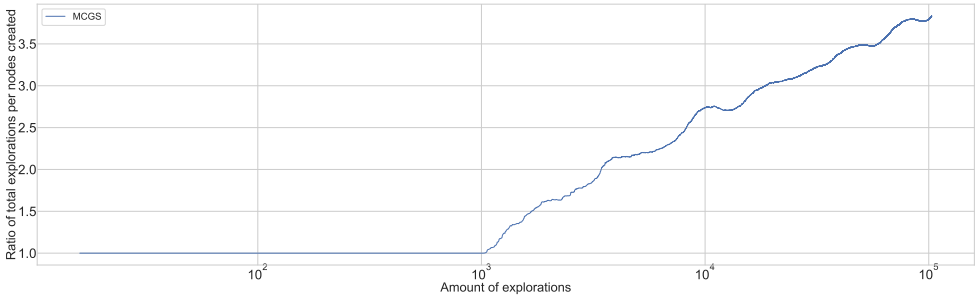



Figure 3.3: **Plot representing the evolution of the ratio of explorations per node created according to the explorations, with MCGS. A higher ratio means less nodes are created during the exploration due to node sharing**

The complexity in time strongly depends on the task, subtask and threads number since most of the computation are part of the simulation phase. The reconstruction of the graph also provides a slight overhead as the graph grows and finding already existing nodes is getting harder. For a graph composed of 262 tasks scheduled (more than 1000 with the subtasks) on 12 threads, between 16 to 20 explorations are performed by the MCGS per 2ms, allowing a total of 8000 to 10000 explorations per seconds. Considering a smaller graph composed of 112 tasks, it performs between 90 and 120 exploration in the same amount of time. We specifically chose 2ms in order to hide this overhead during phases with low parallelism in the task graph by using idle cores (beginning and end of frame). However, the MCGS algorithm is highly parallel as it can explore different nodes and different paths at the same time. The only phase where sequential parts may appear is BackPropagation. One way to handle it would be to update score by batches. The implementation of a parallel algorithm was not fulfilled during the PhD.

Moreover, we provide a way to guarantee a complete control over memory space for the MCGS by defining a "node cleaning system". The number of nodes in the MCGS is limited using a threshold set directly by developers. When the limit is reached, we simply de-allocate node-path with lowest scores and contract all the scoring information in the branching node (the first node of the path) to ensure it will not be explored once more.

## 3.3 Monte Carlo Graph Search for real time scheduling

The Monte Carlo Graph Search method is defined in the same way as MCTS by the following four steps: Selection, Expansion, Simulation and Propagation.

**Selection :** Starting from the root node, the algorithm selects the most promising child node and the path in the graph leading to it depending on the previous explorations of the graph. This selection process continues until it reaches a leaf node (a node without any children). We resort to a modified version of UCT[26][35] to determine the next task to schedule in state $s$:

$$a^* = \arg\max_{a \in A(s)} \left\{ Q(next(s,a)) + C\sqrt{\left(\frac{\ln\left(V(s)\right)}{V(next(s,a))}\right)} \right\} \tag{3.5}$$

where $A(s)$ are the children of node $s$, $V(s)$ the number of visits to node $s$, $C$ a constant, $next(s,a)$ the state reached from $s$ when performing task $a$ and $Q(s)$ the evaluation function defined by:

$$Q(s) = \frac{\sum_{k=0}^{V(s)} \left(1 - \frac{eval(k,s)}{eval_{CG/best}(k)}\right)}{V(s)}$$

$eval(k,s)$ is the evaluation for the node $s$ obtained during the frame $f$ corresponding to the latest $k^{th}$ evaluation of the node, and $eval_{CG/best}(k)$ the evaluation of the best performing scheduling between CG and the MCGS at the moment frame $f$ was tested. This method, depending on the MCGS itself and CG, allows us to maintain already winning strategies for the scheduling without compromising the "safety net" provided by the CG heuristic. Functions $eval$ and $eval_{CG/best}$ are tabulated with circular buffers. In case of multiple tasks having the same evaluation in Equation 3.5, a tie-breaker is to take the lowest indexed tasks with equal score.

**Expansion :** If the selected node is a leaf node, the algorithm expands the node by adding a new child node for each action that can be taken from that state. Each child node represents a different schedule of tasks, where a particular task has been scheduled and its dependencies have been released. In the MCGS algorithm, it is mandatory to ensure that the graph does not contain duplicate nodes, as this can lead to inefficient search and sub-optimal solutions. To avoid creating duplicate nodes, we usually use a data structure such as a map to store all the nodes and their corresponding state. When creating a new node in the graph, we can search the map for a node that represents the same state as the one we want to create. If such a node is found, we can reuse it instead of creating a new one.

**Simulation :** The algorithm performs a simulation for all the nodes created during the previous phase. We determine a scheduling policy starting from the root node, following the path to the expanded node and completing the ordering with CG ordering. We then simulate the execution of the previous frame using this policy in a greedy fashion, using all available resources. This allows the simulation function to generate more informative and meaningful simulations allowing the MCGS algorithm to more accurately estimate the value of a given node and guide the search towards more promising solutions. It is worth noting, however, that the CG heuristic is just one possible heuristic that could be used for this purpose, and other scheduling heuristics may also be effective in certain situations. This simulation is used to estimate the "quality" of the expanded node for a

given frame, allowing to adapt the scoring depending on the most recent executions.

**Propagation :** The algorithm propagates the value of the simulated terminal state in the graph, updating the values of the nodes along the path taken during the simulation. The functions *eval* and $eval_{CG/best}$ are reevaluated accordingly.

This process is then repeated multiple times, with the algorithm choosing different paths through the graph and performing simulations at different nodes, until a certain time budget has been reached.

To highlight the iterative construction of the decision graph explored during the multiple simulations of the MCGS, **Fig.3.4** represents the decision graph depending on the amount of path explored. First, we notice the way the algorithm explores the different available scheduling options from top to bottom. Moreover we also notice that the graph width tends to grow accordingly to task graph's one. We also notice that the use of Coffman-Graham for the completion of the graph provides a way to quickly converge toward known nodes, thus providing a shrink in graph's width when necessary. Finally **Fig.3.3** provides us more information, we can notice that we are still in the phase constructing the "core graph" (<1000 explorations). In **Fig.3.4** We can see that this phase correspond to the first full top down exploration of the graph. the second phase will then provide further variations in the scheduling but will result in the sporadic node creation (following a log(x) node creations ratio with the number of explorations). However, the graph will not stabilize and keep increasing until every possibility is explored, this explains the necessity to provide a way to reduce the amount of nodes dynamically.

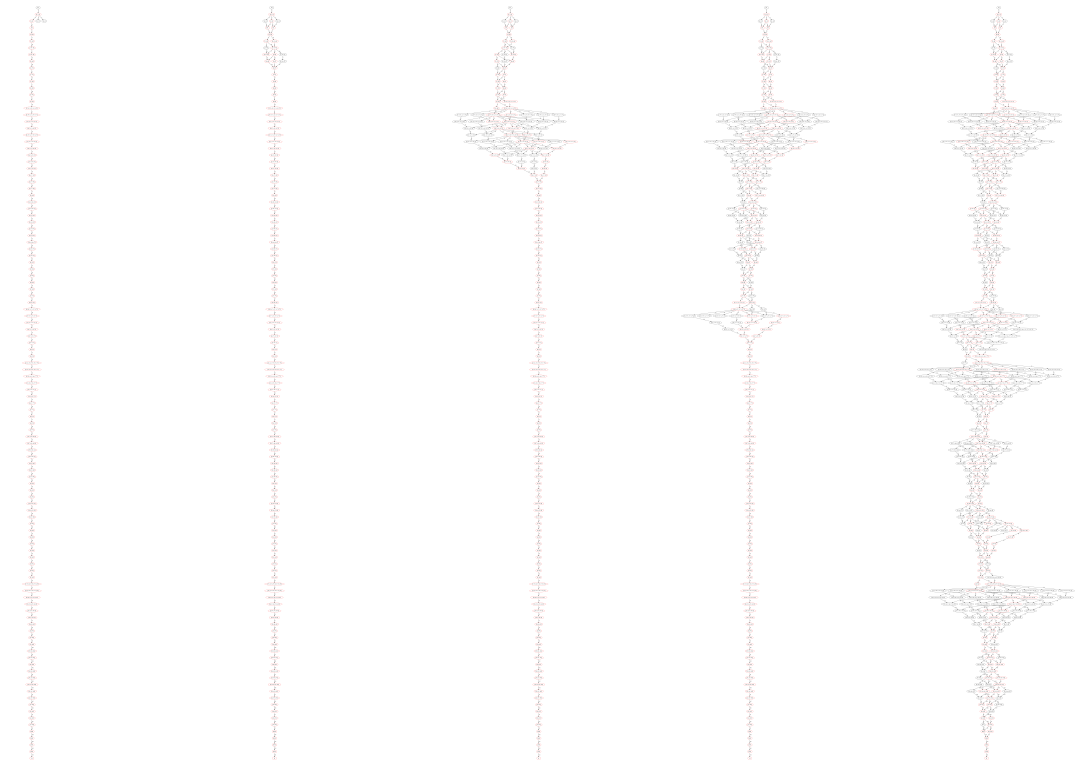
### 3.3.1 MCGS optimizations

In an effort to increase performance, several specificities of the scheduling problem for game engines are used for every step of the MCGS. In our case, the selection not only depends on the previous exploration but is made using different criteria. For instance, if the nodes on the path of the chosen heuristic have not been expanded, they are prioritized, in order to guide the search towards a solution space that seems to be more promising. More complex optimizations are proposed in the following.

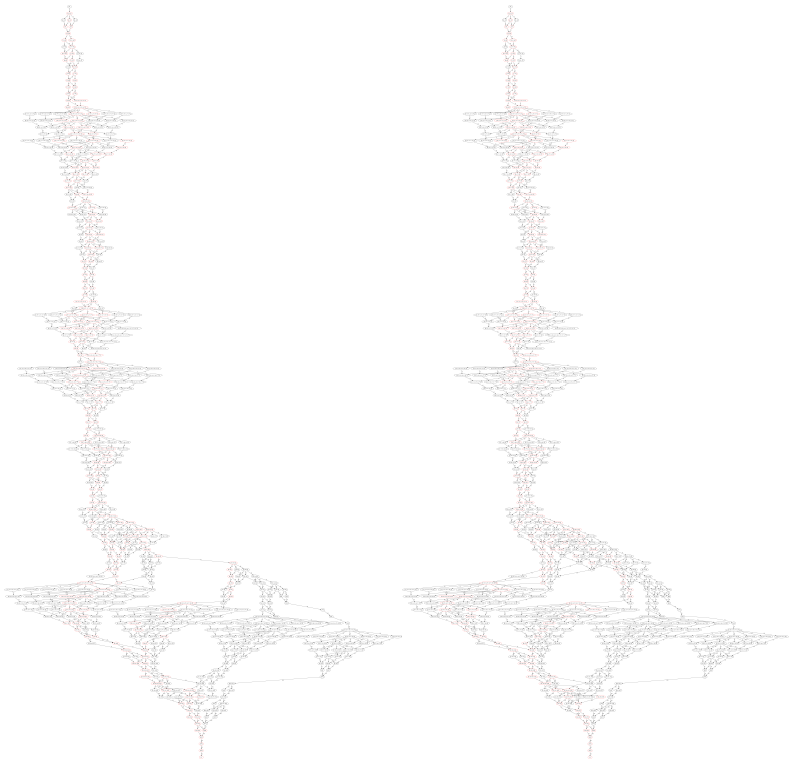
**3.3.1.0.1 Selection optimization** It is possible to limit further the exploration and node creation. During the selection phase, we define the degree $D_f(s)$ of the current MCGS node $s$ at the current frame $f$:

$$D_f(s) = \sum_{t=0}^{|s|} N_f(t)$$

where $|s|$ is the number of tasks in state $s$, $N_f(t)$ the number of subtasks for $t$ (or 1 if $t$ has no subtasks). In this case, if $D_f(s) < P$, the parallelism offered by this node is lower than the number of cores. If all cores were available, changing the priorities between the tasks and subtasks of $s$ would have no impact on the dynamic schedule as they would be all scheduled at the same time. To limit the exploration, we assume that there is no need to further explore in this case. In Equation 3.5 defining the highest priority task, we choose to remove the term counting the number of times this task was chosen, to let the evaluation rely only on CG and possibly another exploration, through another path.



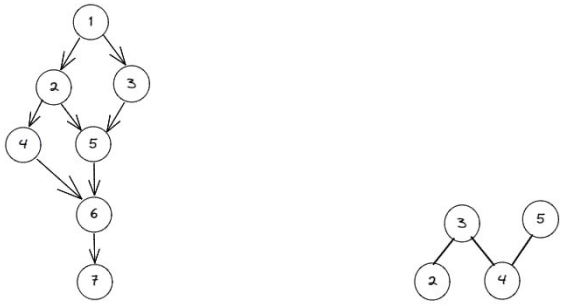
(a) 2 (b) 15 (c) 72 (d) 145 (e) 372



(f) 592 (g) 648

Figure 3.4: **Evolution of the Graph resulting of Monte Carlo Graph Search algorithm depending on the amount of paths explored. One path in the graph correspond to one scheduling.**

**3.3.1.0.2 Expansion optimization** When adding new nodes to the graph, checking for duplication can use the specificities of the MCGS. If the expansion phase is about to add the node $s$ by ordering task $t$, it implies that $t$ is a predecessor of at least one of the tasks in $s$. Similarly, if another path has already built $s$, it implies that a task $u$ is on the edge leading to this node in this other path. $t$ and $u$ are different (otherwise the predecessor state of $s$ in both states would be the same). By construction, only the same set of tasks can lead to the same state, thus $u$ is ordered first and then $t$ in the first path, while the order is reversed in the other path. $u$ and $t$ are parallel tasks. In the task graph, parallel tasks can be interconnected through a graph (an interference graph) described in **Fig.3.5**. For nodes that are reached by a task connected to other tasks in the interference graph, we can register them with the task. When adding a new node $s$ in the expansion phase through the task $t$, we then just have to check that this node has not yet been inserted by one of the tasks independent of $t$.




(a) (b)

Figure 3.5: **Example of task graph (a) and corresponding interference graph (b)**

**3.3.1.0.3 Simulation optimization** To enhances the accuracy of the prediction in the simulation phase, instead of relying on the task duration of the previous frame, a derivative of the time duration is computed for each task and subtask $\delta d_f(t)$ once the timing of the frame $f$ are known:

$$\delta d_f(t) = d_f(t) - d_{f-1}(t)$$

and the duration of the frame $f + 1$ is predicted to be:

$$d_{f+1}(t) = d_f(t) + \delta d_f(t)$$

Instead of using the timing of the task at the previous frame and at the penultimate frame, a larger window could be used.

## 3.4 Experimental Evaluation

To evaluate the heuristic base on the MCGS introduced in this section, several experiments were performed in a controlled environment.

### 3.4.1 Implementation and Methodology

For the evaluation of our methods, we consider two different scenarios based on the task graph described in **Sec.2.2.1**. The first scenario aims to analyze performance and adaptability of the MCGS. It consist in an execution of 200 frames to highlight flaws in classic

heuristics. The load goes from a frame execution below 33ms up to a frame weighing more than 77ms (×2.33) and then goes back to the initial value. The second scenario correspond to an execution of the game on 2000 frames with continuous variation in the load. This simulation runs on 8 threads (instead of 12 initially) to provide scheduling decision with more impact on the execution. Moreover, the load of the game is also augmented but not in the same way as scenario 1. This time the overall span of tasks is increased (load $> 1$) to simulate a game with more interactions with components.
The simulation running both these scenarios was implemented in C++. MCGS runs in real time and is allowed to compute for 2ms per frames on a single core. To enable frame duration comparison, the simulation also runs Coffman-Graham and FIFO heuristics for every frame run by the MCGS.

### 3.4.2 Schedule Comparison

In the first scenario, we respected thoroughly the task graph of game A, the changes in the load are simply due by the increase in span of targeted tasks. The amount of subtasks was not changed. **Fig 3.6** represents the execution resulting of the simulation.

First, we can notice that the changes in the load of the game has a lot of impact regarding Coffman-Graham and FIFO heuristics. These impacts can be seen through the variations in the frame span not occurring on the same frames. However, these impacts are limited as the load of the game increases with a bottleneck between Frames 75 and 125 where the 3 heuristics are very close from one another.

We also noticed that MCGS performs better on the overall execution than FIFO and CG with only 4 frames being worse than both and already providing slight improvements over CG heuristics after 1 frame. After a phase of training (21 frames), MCGS outperforms both algorithms and remains stable even with extreme load variations. Frame 26, it outperforms CG by $\approx$ 4ms and FIFO on frame 174 for instance.

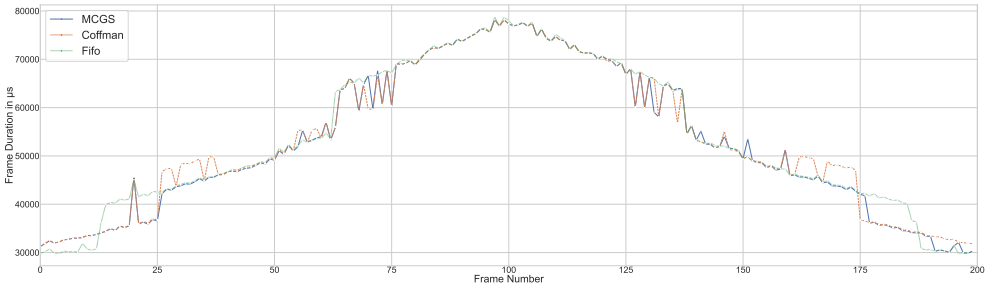



Figure 3.6: **Specific execution comparison of FIFO, CG and MCGS heuristics on 200 frames and simulated for 12 threads on PC with game A**

**Fig.3.7** provides a zoom on Frames 173 to 200. This figure highlights the adaptation capacity of the MCGS algorithm. While before Frame 176, the MCGS seems to be fitting the FIFO algorithm (while slightly improving it), it quickly aligns to CG when it provides better performances. This scenario reoccurs on frame 190 but this time by adapting to FIFO heurisitc. It is important to notice that in this context, MCGS is not aware of the FIFO algorithm but just re-discovers it as soon as it provides better performances.

Another important characteristic of the MCGS is highlighted here, by contrast with other list heuristics. The method is not based on information provided by task graph nor tasks characteristics but only on the performance of the scheduling itself. This induces that in order to provide a worse case example for MCGS scheduling, it needs to be constructed not on tasks specificity but on the method itself. It would require to construct a scheduling loosing performance as we get closer from the optimal scheduling (misguiding the MCGS all the way so it won't explore this particular path).
This induces that MCGS is able to fit any heuristics presented in **Sec.2.4.1** through its explorations and converge rapidly towards it.

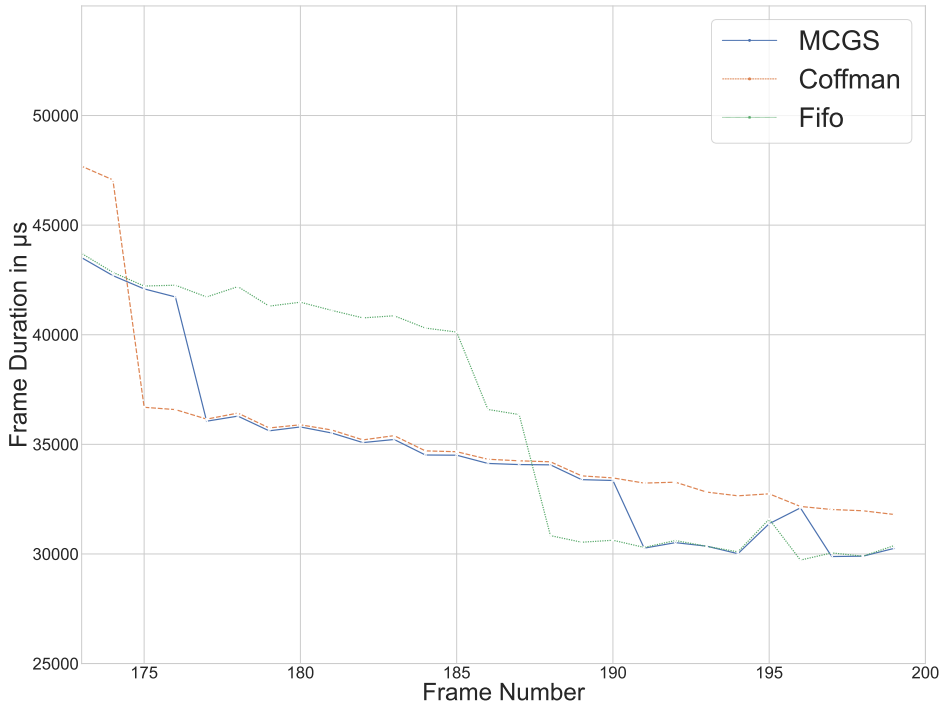



Figure 3.7: **Zoom on frames 175 to 200 for specific execution comparison of FIFO, CG and MCGS heuristics on 200 frames and 12 threads on PC with game A**

The second scenario of our analysis provides additional insight into the potential benefits of using the MCGS on real executions of the game. **Fig.3.8** represents the histogram of frame duration difference with FIFO for CG and MCGS. This plot shows that MCGS tends to perform slightly better than CG. It also exhibits fewer frames with suboptimal performance than the CG algorithm regarding FIFO. Although the MCGS algorithm provides significant performance gains compared to FIFO scheduling, the improvements over the CG algorithm are not as significant. This suggests that while the MCGS algorithm may offer some advantages in specific scenarios, its performance benefits may not always be significant compared to other scheduling algorithms.

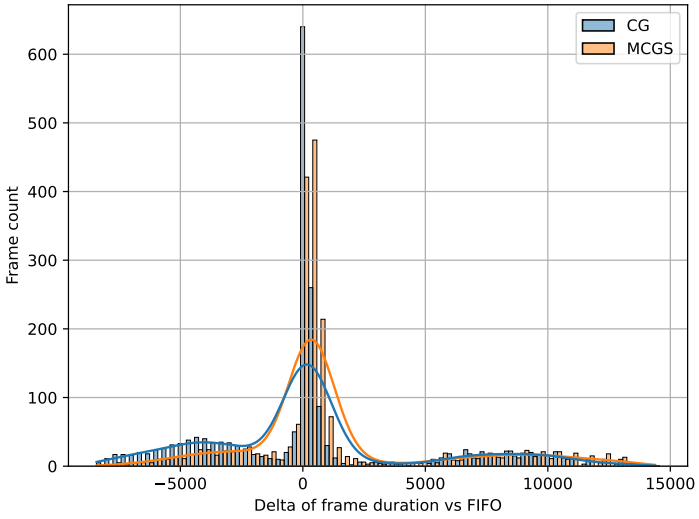



Figure 3.8: **Histogram of frame duration difference with FIFO, for CG and MCGS, on 8 threads for a simulation of game A with PC timings. A positive value means the frame take less time than with FIFO (the is a gain), a negative value means the frame takes longer than with FIFO. The curves shows the density**

## 3.5 Conclusion

These experiments corroborate the capacity of the MCGS to provide an adaptative algorithm depending on the resources available and changes in the load of the game. Moreover this algorithm is time and memory bounded and provides some advantages that will be explored in the next chapter.

Nevertheless, we also noticed that the gain in performances on nowadays engine is limited. This is caused by the lack of parallelism in the task graph considered. This particularity was highlighted in **Chap.2**. Even overloading the engine by increasing the span of tasks was not sufficient to provide substantial gain of performance.

This lack of parallelism is the result of several factors, on the first hand, the engine is developed by a great number of developer on a large period (several years). Moreover, these developments are often occurring during the game implementation. During this period, engine may be twisted and hot fix. It induces a complexity to track down and document all data dependencies. On the other hand, as the engine is constructed and updated iteratively a co-evolution between engine and hardware is introduced.

To provide game engines and task graph more parallel, we need to be able to free the game from this co-evolution. This process was already made possible for GPU tasks through the introduction of level of quality for graphic computations. One solution would be to develop similar level of quality system regarding CPU tasks in the engine. The complexity emerging from this problem is to limit the disturbance in the execution of the game while providing maximum performance.

# Chapter 4

# Time Skip and Skippable Tasks

We defined and evaluated in previous chapters different scheduling techniques, some from the state of the art, some specifically designed for video games providing performance improvement. However even though scheduling tends to improve the execution of the game by providing a better anticipation of computations and a better load balancing, we noticed that these methods alone are not sufficient.
As introduced in **Sec.1.3** a certain co-dependency exists between the content of the game (thus its execution) and the platforms the game is supposed to run. This co-dependency is curbing the versions for newest generation of consoles and high performing PC. It was described as resulting from an impossibility to define general quality level for logic treatment in the game such as AI, physics or audio as it would change noticeably gameplay thus the game.
However, to create a Level of Detail's system on CPU tasks as currently implemented on GPU ones. One solution is to dynamically postpone sporadically non-mandatory treatments in the game. This deferment is to happen when it seems these tasks may induce frame drop or an overload to balance computations on several frames, avoiding over computations on a single one.

## 4.1 Time Skip and Skippable Tasks in Game Engines

This method is based on Skippable tasks. This particular set of tasks are not preemptible, neither moldable nor malleable as in Marchal et al [31] and, they cannot be cancelled, but they can be slightly postponed without compromising the integrity of the system nor inducing frame drops. These skippable tasks have already been introduced in different works [42, 11, 27, 33] but these works focus on independent jobs and cannot be applied here. In the context of video games, subtasks are included in the task graph and defined by the game developers. These tasks are various, from AI behavior planning to explosion management, physics computations or loadings.
Usually, these treatments are handled in two different ways, depending on their integration in the task graph. For instance, AI behavior and explosions are included in the task graph and events will trigger these tasks at certain point depending on the state of the game. Sometimes these triggers depend on frequency pattern (ex: re-compute path finding every 20 frames). When considering these tasks with a periodicity of 20 frames, we don't take full advantages of their "skippable" specificity. Moreover, in some cases, a certain synchronization of such tasks may occur, resulting in a significant overloading for the frame in comparison to regular one. For instance, when dealing with 50 NPCs, if

all paths findings are computed on the same frame, this may results in major frame drop while managing 5 at a time would have been fairly smooth.
For tasks not included in the task graph such as loadings, we take advantages of OS's scheduler by handling them on a separate thread with low priority that will preempt a core from time to time to execute its instructions. However this method causes issues as well since the absence of control on the execution may induce massive frame drop in order to catch up accumulated computations.

We propose here to handle these tasks in a different way by defining an adjustable bound(Skip-index) for task's execution. This allows skippable tasks to be computed on a range of frames. Using this skip-index, we provide the possibility in some cases to slightly postpone computations when the engine is not in capacity to compute the frame. This results into the distribution of computations on a greater amount of frames.

**Fig.4.1** shows an example of massive NPC management on the game Ultimate Epic Battle Simulator. In this context, computing path finding or behavior for every AI displayed on the screen even if simplified is way too intensive in term of computations.

Figure 4.1: **Illustration of extreme NPC handling in Ultimate Epic Battle Simulator**

The main issue with this method is to determine in advance the amount of available computation capacities and determine the amount of skippable subtasks that should be onboarded for a given frame. This requires the capacity to anticipate incoming computations, and even provide hints on how scheduling would be impacted if such tasks were to be executed on this frame.
The method we introduced in **Chapter.3** provides a scheduling oracle every frame using a MCGS.

## 4.2 Dynamic Time Skipping using MCGS

Skippable tasks are tagged by the developer and only tasks with subtasks can be tagged as skippable. This induces that skippable tasks cannot be skipped directly but all its subtasks can. However, we notice only a fraction of the subtasks are skipped when the load is too high and not the totality as they can be spread on $Skip - index + 1$ frames.

The different skippable tasks composing the frame are tagged with a skip factor. This defines the maximum delay the subtasks composing these tasks can undergo. We now extend this notion to skippable subgraphs: a skippable subgraph $S \subset G$ is a subset of skippable tasks, where all tasks, at any given frame are linked as they are co-dependent. This means that if one of this subtasks is executed or skipped the other subtasks composing the subgraph should be handled in the same way for this frame. All the subtasks composing the subgraph are co-dependent as they aim to update similar components of the engine. These subtasks represent parallel for iteration on an array of component (for instance having 42 NPC will create 42 subtasks for every AI updates tasks). In this context all the tasks composing the skippable subgraph have the same amount of subtasks. This number of subtasks is defined, per frame, for the whole subgraph $S$. Considering two tasks $u$ and $t$ in $S$, if $u < t$, the subtasks of $u$ only have a link with the corresponding subtasks in $t$, meaning that if the $u^i$ is skipped then $t^i$ must be skipped as well. A skip factor is defined for the subgraph $S$, and multiple disjoint skippable subgraphs can be defined with different skip factors each. This value is set by game developers to guarantee coherence of the game. In the following sections we consider various skip index to evaluate the impact it provide on frame executions.

As we explore the MCGS to improve current scheduling, the simulation phase provides a fast simulation of the scheduling to enhance its performance iteratively. This simulation provides a lot of information on the behavior expected by the upcoming frame. In this context, the MCGS algorithm can pin down the amount of required computations for a given frame and anticipate if the next frame will not meet the pre-determined threshold (33.3ms for 30 Frame per seconds, 16.6ms for 60 Frames per seconds).
If the scheduling heuristic notices that the frame will not be computed in time with all planned skippable tasks to be computed, the algorithm performs simulations with a certain amount of computation and determine the fine tuning of skippable tasks allowing the task to respect the required FPS. This mechanism is illustrated in **Fig.4.2**.

In order to determine this fine tuning, we consider all the skippable subtasks available (with a remaining skippable limit stricly superior to 1) as an ordered list. This list follows an Earliest Deadline First order, where the deadline is represented by the amount of frames remaining before forcing the execution of the subtask. The algorithm then performs simulations including only a subset of the subtasks in order to determine as precisely as possible the amount it can board for the next frame. We proceed in a dichotomic search on the subtasks list, trying to board half of it then depending on the result of the simulation considering 3/4 of the subtasks if the frame was respecting the pre-determined limit or 1/4 of the subtasks if it was not.
Proceeding this way provides better performances as it requires $\mathcal{O}log_2(N_{st})$ simulations in order to fine tune completely the amount of subtasks that should be computed for a given frame.

Moreover, to provide maximized performance, we extend the notion of derivative

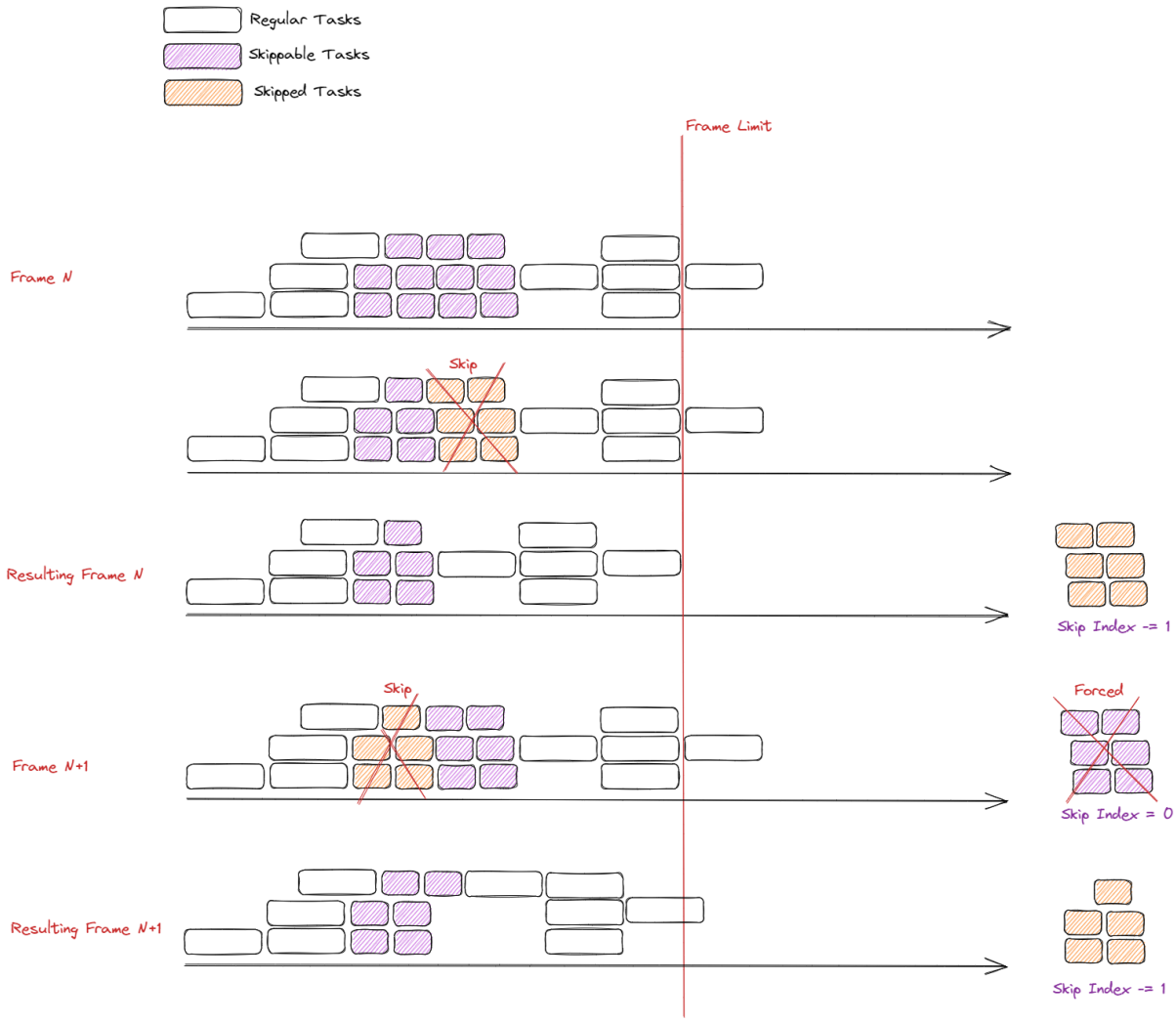



Figure 4.2: **Illustration of frame scheduling with task skipping mechanism on 3 cores**

frame. In previous chapters, simulation's estimations for incoming frames were initially based on the previous frame. We then introduced the derivative frame taking into account not only previous frame but also the variations on several frames to have a more precise estimations and thus anticipate incoming variations providing a better adaptation for the scheduler.
We extended this derivative frame, by taking into account skippable tasks and their remaining skip index to provide more information to handle the load to board on a given frame. As shown in **Fig.4.3**, using the derivative frame instead of the previous one resulted in a better anticipation of incoming computations for a given frame. It induces a significant lowering in the amount of frame drop, while remaining fairly close to the frame limit. However, we noticed this method tends to be more cautious regarding the amount of tasks onboarded, resulting sometimes in slightly shorter frames than it should with an optimal method.

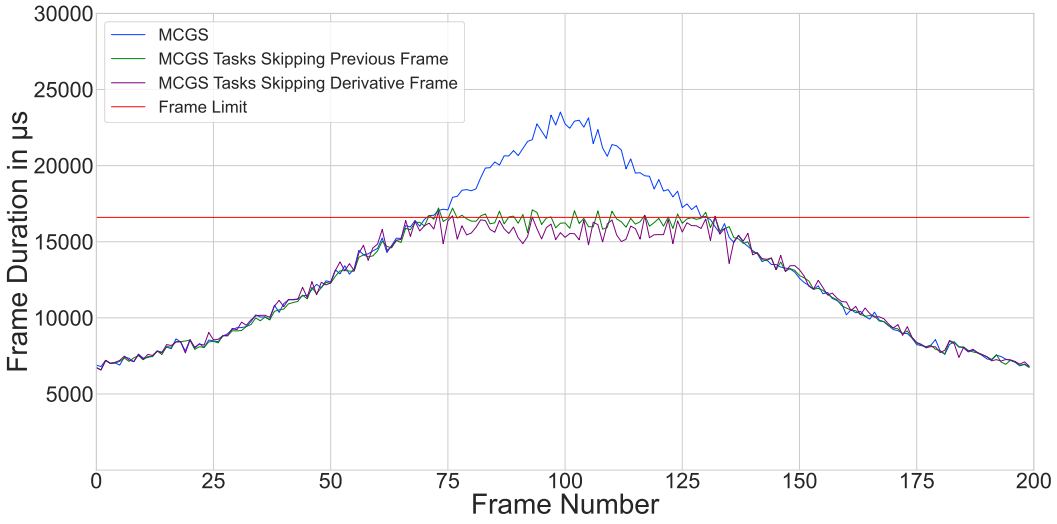



Figure 4.3: **Execution comparison of MCGS based on Derivative frame and MCGS based on Previous frame on 200 frames and simulated for 12 threads on PC with game A**

## 4.3 Experimental methodology

We propose to study two recent AAA games from Ubisoft to evaluate our different experiments. These games originated from two different versions of a same engine that diverged several years ago. Computations requirement are different depending on the game due to their different types.

| Name | Type | Year |
|---|---|---|
| Game A | First Person Shooter | 2022 |
| Game B | Third Person Action Aventure | - |

Table 4.1: **Video games studied**

### 4.3.1 Implementation and Methodology

To assess the impact of skippable tasks and of the skip factor, we propose 2 scenarios:

- In the first one, we consider **a continuous load variation over 2000 frames due to a varying number and load of AI tasks**, for game A on the PC architecture. The load variation follows the same pattern as in **Sec.2.3.2** on a larger amount of frames and representing 2 cycles of growth and shrink of computations in the engine. This aims to provide a better representation of computation's variations in the engine during changes of games phases during real executions. Increasing the load of the frame impacts directly the amount of AI considered and the amount of subtasks composing its update. This directly affect tasks considered as skippable but not exclusively as it would normally in a real execution. This experiment allows us to have more control on the evolution of execution allowing to notice behavior changes of the algorithm with extreme variations in load management.

- The second scenario is focused on game B and is very different as it is not simulating frames based on extracted information on profiled frames but on real continuous executions extracted from Xbox One and PS4 Pro in-game profiling. The method used to reconstruct the task graph of the game is detailed in **Sec.4.3.2**. We propose in this scenario two different experiments:
  the first one is classic task skipping for the two platform and it's impact on the video game and it's perceived frame rate (also called visual frame rate) for the user. The second experiment aims to simulate disturbance in the execution such as asset loading in the video game (OS pre-emption would have similar effects). **To simulate the load of assets, one core is removed from the 8 cores normally available to compute the frames for some time interval**, and when the loading is done, the core returns to the execution of the frames. In both these experiment, only one AI skippable task is considered.

| Name | CPU | GPU |
|---|---|---|
| PS4 Pro | 2.1Ghz 8-core AMD "Jaguar" | AMD Neo |
| Xbox One | 1.75Ghz 8-core AMD "Jaguar" | AMD Durango |
| PC | 3.6GHz 6-core Intel E5-1650V4 | Nvidia 1080Ti |

Table 4.2: **Characteristics of platforms considered for the experiments**

**Table.4.2** describes the different platforms taken into account for both scenarios. We decided not to choose PS5 nor Xbox Series X (even though it would have been possible) because the current transition phase where PS4 and PS5 (resp Xbox One and Xbox series X) co-exists directly impacts the video games released as they are tailored for 8th generation of consoles. This induces that the amount of CPU computations are adjusted to the worse performing device. Resulting in under used CPU for newest generations with frames lasting roughly 11ms where oldest generation are most of the time slightly overload.

### 4.3.2 Automatic reconstruction of Task Graph

To be able to simulate the execution of the engine on Game B, it is required to be able to replay the different tasks captured. This implies being able to reconstruct the task graph for game B. As described in **Sec.2.2.1** extracting the task graph of the game is complex and requires to manually track the tasks in code. However, techniques that were not available in early years of the PhD allows us now to simplify this step.

Using profiling tools and capture of the game, we are able to track for a given frame the task triggering, on completion, the release of the last dependency for its child resulting in its scheduling. We define such relation between two tasks as a "link". By sampling a high quantity of frames, in various contexts, we are able to recreate all the different links between tasks. Then by removing loops (due to iterative execution on array) or transitive dependencies, we are able to reconstruct a task graph based on real executions without any knowledge of the code.

One of the downside of this method we must denote here is that the resulting task graph does not describe real data dependencies but implicit ones described in the code by

developers, moreover even though it strongly limits the amount of work to create such task graph, some additional treatment based on knowledge of video game engine is necessary to finalize dependencies (for instance between Graphic and Engine loop). **Fig.4.4** (b) represents the task graph obtained with such method on video game B.

## 4.4 Experimental Results

The experimental evaluations of the algorithm were performed in similar conditions as MCGS evaluation described in **Sec.3.4.1**. The simulation was implemented in C++. MCGS runs in real time and is allowed to compute for 2ms per frames on a single core. To enable frame duration comparison, the simulation also runs Coffman-Graham and FIFO heuristics for every frame run by the MCGS.

### 4.4.1 Dynamic adaptation of AI updates

For the first scenario, we consider one AI skippable task and various skip factors (1, 2, 5, 10, 15). **Fig 4.5** represents the evolution of frame duration during the run. **Fig 4.6** a-d show the evolution of the AI Update frequency with and without the task skipping. and **Fig 4.6** e-h the correspondence between frame duration with and without task skipping. The colors in these figures describe the same frames in plots. Green dots represent frames with a computation time under 16.6ms, orange are the frames that do not cope with this threshold but manage to remain globally stable when task skipping is activated and Red are the frames unable to meet the deadline even with task skipping.

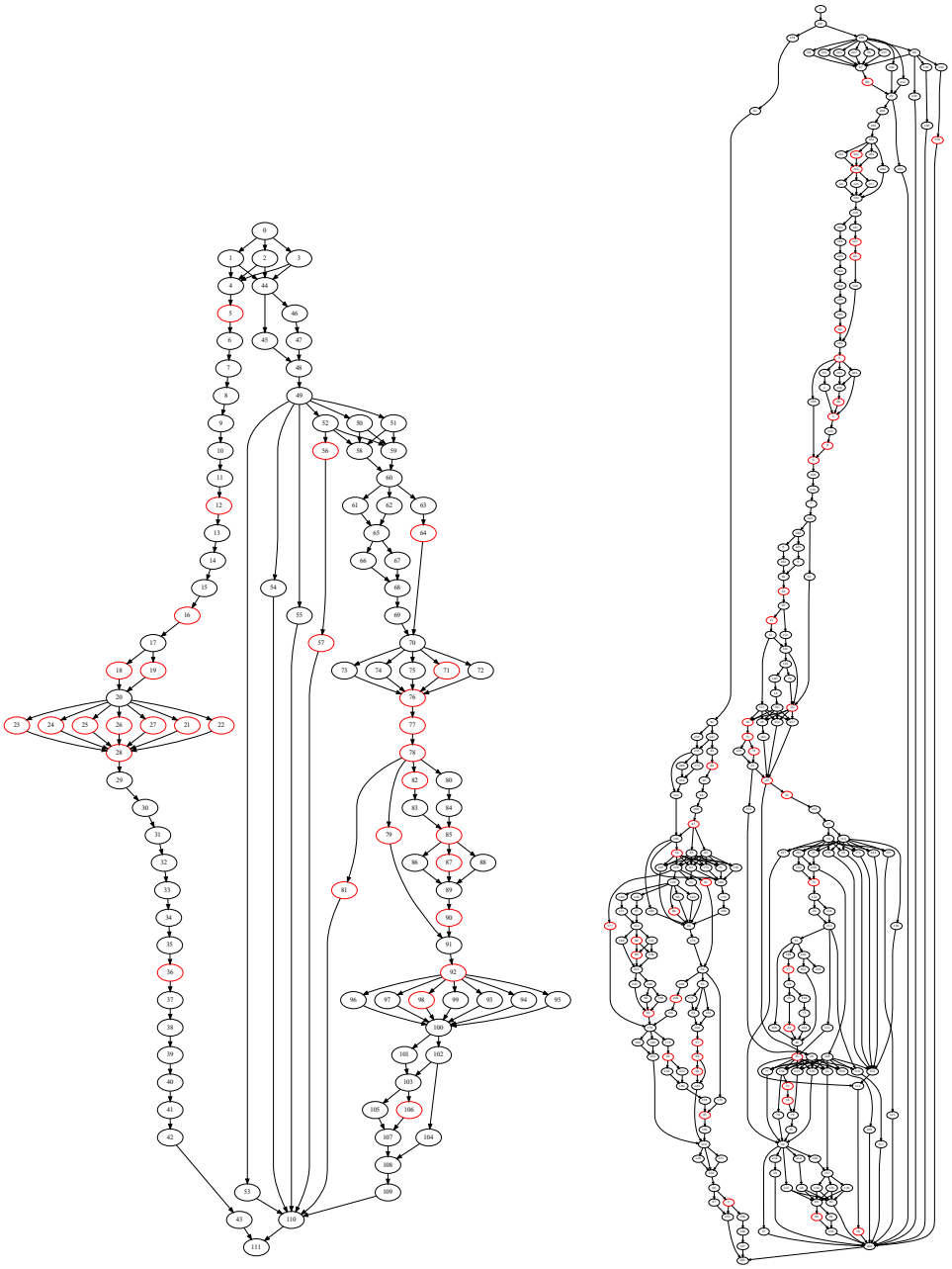
Regarding the frame duration, **Fig.4.5** illustrates algorithm capacity to provide a certain robustness for frames that would have normally been dropped.

In Figure **Fig.4.5**, we notice two different behaviors, the first one occurs when the required computations is limited regarding skip index **Fig.4.5 (a)**. The algorithm reduce strongly the amount of computations, enabling to meet the deadlines in most of the studied cases. For instance on the rise and fall simulation (Frame number 0 - 1000), a skip index larger or equal to 5 enables to avoid most of the frame drop without causing any computation accumulation. Moreover, for an inferior skip-index, even though some frames are dropped for the most computation intensive phases (Frame number 420 - 565) frames duration is fairly reduced. We can also notice a split in the amount of tasks with a fair repartition occurring, for a task $u$, $\frac{N_{st}(u)}{SkipIndex+1}$ subtasks are scheduled every frames if subtasks are relatively equal in terms of span.

The second behavior occurs when the skip index is not sufficient and, the frame threshold to respect in comparison to its real duration is too high **Fig.4.5 (b)**. In this context, an oscillatory state takes place, inducing strong variations in computations with a frequency of $\frac{1}{SkipIndex+1}$.

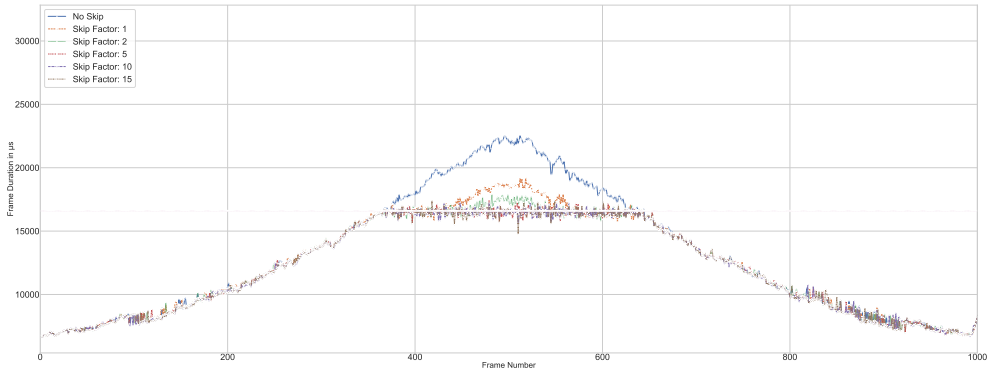
This behavior can be explained by the lack of skippable tasks and the impossibility for the algorithm to respect the deadline, even when skipping most or totality of it. All these tasks are then reported until the limit is reached resulting in the scheduling of all the remaining tasks.

**Fig.4.6 (e), (f), (g), (h)** describe with further detail the behavior of the different frames composing the execution represented in **Fig. 4.5**. These plots show in detail the comparison between frames with and without task skip. We can notice that the algorithm enables, for frames that would have normally dropped(Yellow and Red), to meet

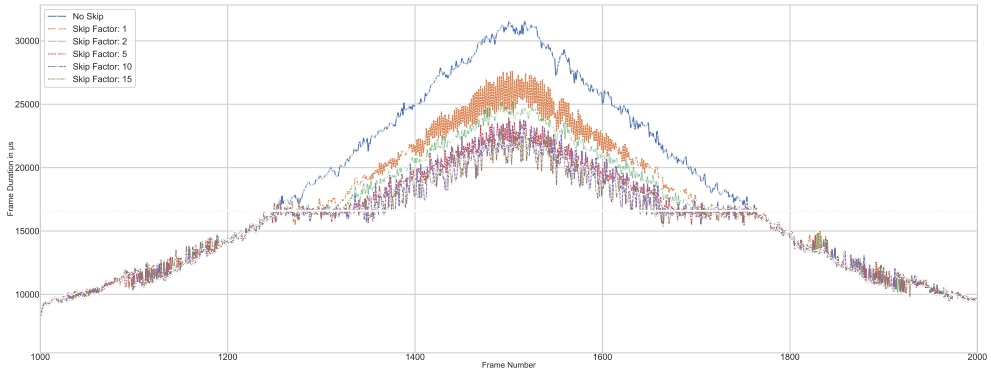


(a) (b)

Figure 4.4: **Graph extracted using profiling resources in Ubisoft, (a) represents the task graph for Game A extracted by hand and (b) the task graph of Game B extracted automatically**




(a)




(b)

Figure 4.5: **Comparison of MCGS scheduling with and without task skipping, on 12 threads simulated on PC for game A. (a) represents small overload of the frames while (b) represents large variations to highlight tasks skipping behavior on edge cases**

the deadline of 16.6ms. **Frames executing up to** 19 **ms can be executed in** 16.6 **ms with a skip factor of** 1**, up to 23 ms with a skip factor of 5** and over. It also provides a slope below 1 for the frames in red, amortizing the frame duration increase thanks to skipping. Having a skip factor of 5 greatly extends the transition phase (from 19ms to 23ms) and frame duration slow increase (from 0.8 to 0.6) for the red frames. Increasing the skip factor to 15 does not change those two values but shows a lot more dispersion on frame duration, allowing in some cases to fulfill the required time per frame in a sporadic way. We can also notice the two different behaviors described previously in plot (e) through the splitting occurring for frames with the longest duration (over 25ms).

Regarding the amount of AI updates per seconds shown in **Fig 4.6**.a-d, we notice the same behavior as in the previous plot. Task skipping occurs for orange and red frames. The behavior is transient for orange frames and the frame duration still corresponds to approx. 16.6ms. For the red frames, while their duration exceeds the threshold value, the task-skip is maximal (subtasks are scheduled only when forced to). Nevertheless, This degradation respects also the requirement on updates per seconds even though it is not

ensured by the algorithm itself (the skip factor ensures at least one execution per frames).

The second scenario, as the previous one, only consider one AI skippable tasks and, proposes different skip-index (1, 2, 5, 10, 15). Frame and tasks duration are extracted from in-house continuous tests of the game B and represent the same gameplay phase. Theses tasks are profiled and replayed following the task graph in **Fig 4.4 (b)**, targeting a frame duration of 33.33ms (30 FPS). It consists in real execution extracted from Game B on both Xbox One and PS4 Pro.

First of all, we studied in **Table .4.3** the impact of task skipping on the visual frame. The visual frame rate refers to the number of images displayed on the screen each second. To calculate this rate, we need to consider both the screen's refresh rate and the duration of each frame, as we must determine when the screen synchronizes with the frame buffer being displayed. For example, if a screen has a refresh rate of 60Hz, even if the engine processes 240 frames per second, only 60 frames will actually be shown on the screen. We also consider Vertical Synchronization (V-sync). It ensures that only whole frames are shown on-screen, preventing screen tearing by waiting for the screen to synchronize before displaying a new frame. When a frame misses the deadline for synchronization, the engine waits for the next cycle, resulting in a longer visual frame duration than the actual frame computed by the engine. Suppose a frame is expected to be processed in 16.6ms, but it takes 20ms, causing it to miss the synchronization between the screen and the image buffer. Instead of starting a new frame, the engine will wait for the screen to synchronize, avoiding overwriting the current image buffer. In this scenario, even though the actual frame duration was 20ms, the visual frame captured on the screen will have a duration of 33.3ms.

Table 4.3 shows the different figures for screens from 60Hz up to 360Hz. The results demonstrate that on Xbox, **the number of visual FPS increases slightly with the frequency of the screen and MCGS with skip factor allows a gain of up to** 12.8**% of framerate (**+4 **FPS)on Xbox,** 14.3**% of framerate (**+4.57 **FPS) on PS4**. Moreover, even though these optimizations cannot guarantee a stable 30FPS all the time, the solution takes place by skipping only one AI Task. This method can be applied to various tasks and skippable subgraphs in the engine providing even more gain and stability in performance.

| Screen Freq. | Xbox Real | Xbox MCGS | Xbox MCGS-1 | Xbox MCGS-5 | Xbox MCGS-15 | PS4 Real | PS4 MCGS | PS4 MCGS-1 | PS4 MCGS-5 | PS4 MCGS-15 |
|---|---|---|---|---|---|---|---|---|---|---|
| 60Hz | 28.17 | 27.50 | 28.83 | 29.59 | 29.66 | 28.80 | 28.18 | 29.13 | 29.66 | 29.83 |
| 120Hz | 29.00 | 28.71 | 29.61 | 30.81 | 31.59 | 29.40 | 29.10 | 30.58 | 32.29 | 33.62 |
| 240Hz | 30.55 | 30.13 | 31.59 | 33.61 | 34.48 | 31.46 | 31.17 | 32.81 | 34.89 | 36.10 |
| 360Hz | 31.36 | 30.78 | 32.37 | 34.40 | 35.38 | 32.44 | 31.93 | 33.63 | 35.73 | 37.01 |

Table 4.3: **Visual FPS on Xbox and PS4 Pro on real execution and MCGS with 1,5, 15 skip factors, according to screen frequency.**

Figure 4.7 shows for both architectures a cumulative percentage of frames that execute in less than a given duration (in the x axis). For instance, it shows that **on PS4 Pro (top figure), nearly 0% of the real frames are executed in less than** 25**ms (40 FPS), while 20% of the frames meet this deadline for MCGS and time skipping with a skip factor of 2, 30% for a skip factor of 5 and 40% for a skip factor of 10**. We notice that we tend to slightly over-estimate the duration of frames in comparison to the real execution (comparing the MCGS with no task skipping and the real execution). This might be caused by some slight inaccuracy in the task graph or in the method used

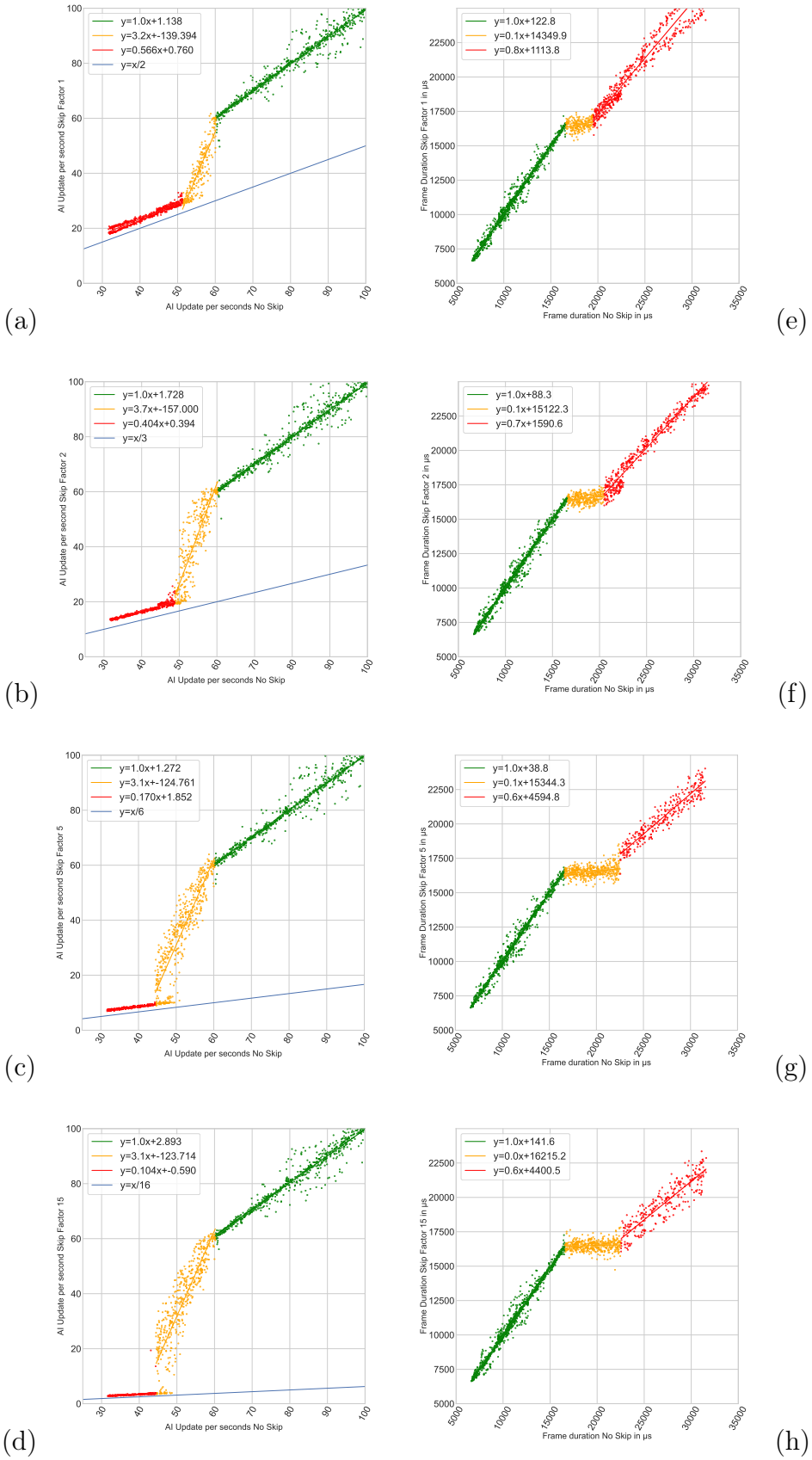



Figure 4.6: **Simulated executions on 12 threads with game A. On left, AI update per second with task skip. On the right, frame duration with task skip. From top to bottom, Task skip index of 1, 2, 5, 10, 15. Colors red/orange/green identify the different zones and correspond to the same frames in plots in the same row**

by the profiler to compute the duration of the frame itself. It also shows the impact on frame duration of task skipping on both PS4 and Xbox One, with **an increase of 15% in the number of frame computed below 33.33ms on PS4 and 17% on Xbox**. By applying a task skip of 1, the impact is significant, gaining more than 10% of the frames below the threshold on both platforms. It also shows significant improvements on frame duration under 25ms, increasing the number of frame below this value up to 40% on PS4 and 20% on Xbox which makes a lot of difference as it represents the limit for 45FPS.
We can also notice that the skip factor has a diminishing return in terms of performance. This is due to several factors: skipping subtasks reduces the amount of parallelism available. Besides, using a skip factor of $n$ roughly divides the number of subtasks by $n$ (assuming a constant load). Thus changing a skip factor from $n$ to $n+1$ brings higher time reduction than from a skip factor of $n+1$ to $n+2$.

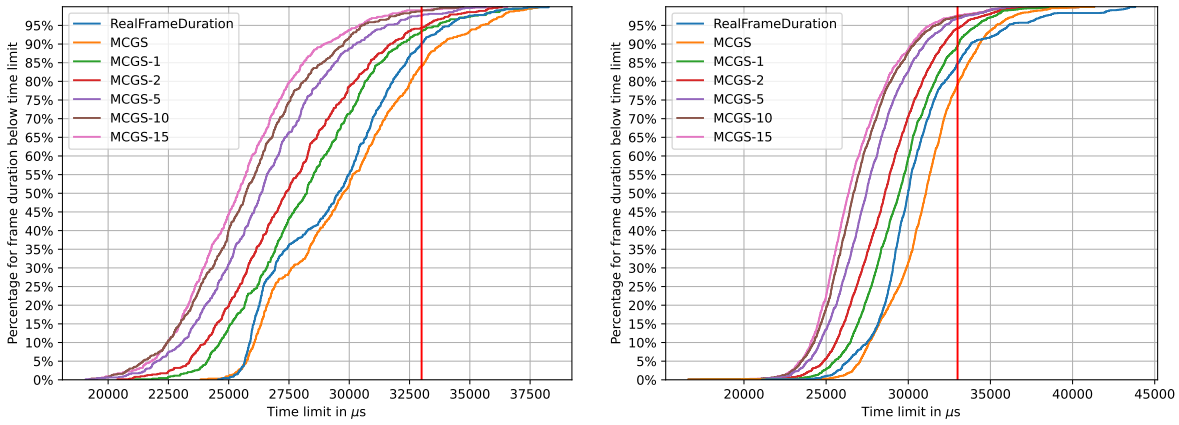



Figure 4.7: **Comparison of real execution with various skip factors on PS4 Pro(left) and Xbox One(right) representing the percentage of frame duration below time limit in µs. RealFrameDuration represents the duration profiled in game while MCGS represents the replay of the frame with tasks profiled obtained by simulation.**

### 4.4.2 Load Management

Frame drop can occur in a video game for various reasons. It can be caused as evaluated in the first experiment through load variation in the game. Another reason is thread preemption or loadings occurring in the game and diminishing the amount of resources available for a short lapse of time. The second experiment aims to evaluate Time Skipping performance in such circumstances. For the second experiment, we compare the frame duration of CG using all 8 cores (no loading), of CG using only 7 cores (1 core loads data) and of MCGS when the load occurs, with task skipping enables with a skip factor of 15. On this machine, the target framerate is 30 FPS (33.3ms per frame). The session played starts a few frames before the load occurs and stops a few frames after. This clearly shows that on CG, when the load occurs, most of the frames that were executing in less than 33.3ms execute now in more than this threshold, causing frame dropping. When the thread preemption does not occur, CG is able to ensure that 83% of the frames execute in less than 33.3ms. When the loading event takes place, this drops to 28% with

CG, while **MCGS and task skipping are able to schedule 66% of the frames below 33.3ms during the asset loading event, automatically**. This represents a decrease of 38.5% frames drops in comparison to the results obtained with CG. This second scenario illustrates that the automatic task skipping is able to smoothly degrade the quality of skippable tasks even in the case of a fast change in workload in order to guarantee frames drop will occur as little as possible.

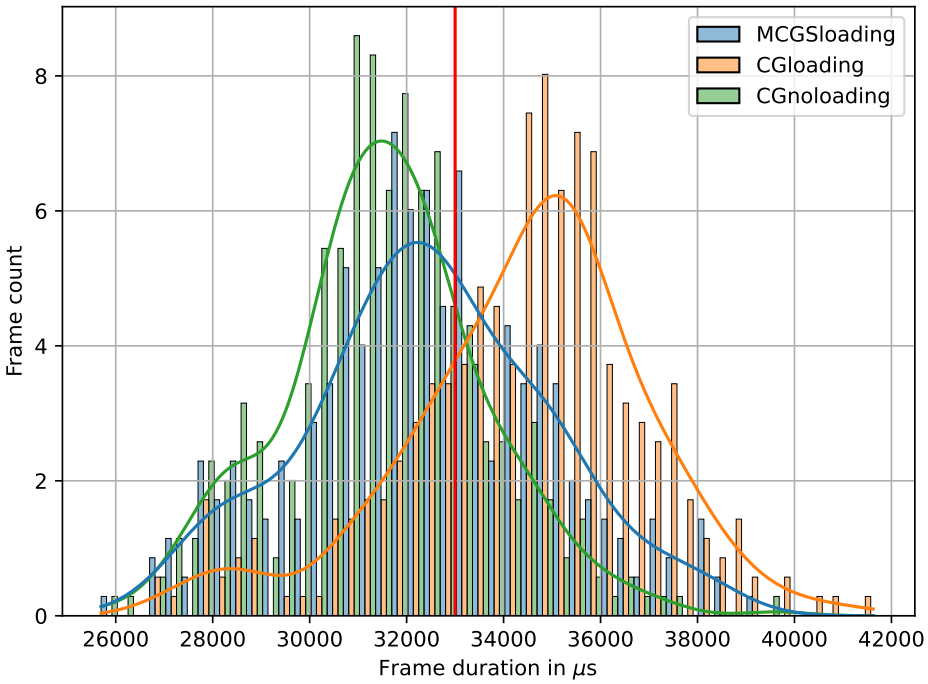



Figure 4.8: **Histogram of frame duration comparing MCGS with a skip factor of 15 and CG when an I/O occurs (taking 1 core out), and CG without I/O for a simulation of game B with Xbox One timings. The curbes shows the density.**

## 4.5 Conclusion

The outcomes of these experiments highlight the potential to enhance video game performance by optimizing scheduling and execution of the tasks composing the game engine. This approach can be regarded as a Dynamic Level of Quality for sporadic CPU tasks. For example, by reducing the update frequency, as illustrated in **Fig.4.6 (a)-(d)**, the AI's responsiveness decreases. However, this trade-off enables the possibility to significantly increase the number of AI characters that a video game can manage synchronously, as well as improved performance on devices with lower amount of resources. This adaptability allows for AI, explosions, and loading quality to be adjusted based on the game platform's architecture and available resources at any given moment, ensuring tailored execution to specific hardware.

An alternative implementation of this solution may involve using a simplified version of AI tasks, explosions, or loading, instead of delaying them. This approach would lead to a more advanced dynamic level of quality for CPU tasks, accommodating a broader range of hardware configurations and avoiding the reduction of responsiveness by providing other

methods of computations. While this method would necessitate a more intricate implementation to accommodate various task versions, the resulting flexibility would benefit both game developers and players.

# Chapter 5

# Conclusion

This PhD focus on the problem of scheduling in video games engines.

When developing a video game, a set of tasks must be executed on a player's machine at each frame, such as the physics engine, sound management, and artificial intelligence. Each of these tasks can be expressed in a monolithic way or decomposed into a small parallel graph. They are grouped together as a global graph and must be executed efficiently to guarantee a target frame rate. Moreover, it is necessary to respect the dependencies between tasks and their priorities in order to maintain both coherence during game execution and to make the best use of available time. One initial difficulty lies in the fact that these tasks are built and added by different individuals, whose professions are independent. The global task graph is obtained through composition and is very difficult to access during the video game design process. In addition, the number of tasks and their execution duration varies throughout the course of the game depending on the player's position, actions, environment, or other players. These variations in tasks duration also induces a variations in the critical path of the frame, impacting directly the performance of the scheduling method chosen.

Another challenge in optimizing the execution of tasks in a video game is the inability to produce "level of quality" for CPU tasks. Unlike GPU calculations, which mainly focus on the visual side of the game, CPU calculations directly modify the state of the game and its proper functioning. Therefore, any changes or optimizations made to CPU tasks must be carefully implemented to ensure that they do not negatively impact the overall gameplay experience. Moreover, CPU tasks can be highly dependent on the game's design and logic, which can make it difficult to generalize optimization techniques across different games. For example, the way in which AI tasks are executed and integrated into a game can vary widely, depending on the game's genre and mechanics.

## 5.1 Contributions

In this PhD we proposed a definition and a formalization of the model of task scheduling for video games as well as metrics to measure performance in video games. Using these metrics, we extensively studied state from the art scheduling techniques to provide insight into their effectiveness on this particular problem. To achieve this, we proposed the creation of a simulator to generate representative frames or replay real ones extracted from the game engine, allowing for the estimation of scheduling performance and frame

duration. These simulations highlighted the possible gains in performance that can be achieved by using scheduling heurisitcs. Moreover we proposed several techniques to improve performance : Multi level scheduling for parallel subtasks and targeted tasks parallelization.

In order to take into consideration the different evolutions video games could face in the next years. We introduced and evaluated an adaptative scheduling method bounded in space and time based on the Monte Carlo Graph Search algorithm. These evaluations highlighted the capacity of the heuristic to adapt the scheduling to the amount of computations or resources available. Using the oracle capacity of the MCGS, we introduced a method enabling level of quality for CPU tasks in the game engine. The Task skipping method provides the capacity to postpone some part of the computations on next frames until the threshold settled by game developers is hit. Through the various evaluation of this method, we highlighted its capacity to provide stability for the frame rate in the context of computation's overload or limited resources.

## 5.2 Perspectives

While we have covered many aspects of the scheduling problem for video game engines, interesting directions remains to be to explore. First of all, the implementation of the different research conducted in this PhD in a production engine of Ubisoft. To provide the engine with such mechanism, we need to be able to guarantee a strong control over tasks and their data dependencies. One idea would be to introduce a system of input/output declaration for tasks enabling the reconstruction of the task graph as defined in [7].

Another interesting perspective would be to improve the MCGS and Time Skipping method. One way would be to use machine learning algorithms. These algorithm could be applied in various stage of the method, for instance instead of using a derivative frame as an oracle, a deep neural network could be the one producing the expected frame. The simulation phase could be positively impacted as well by replacing the fast evaluation of the scheduling by a probability of having a better scheduling as implemented in AlphaGo.

Another way to enhance performances of the MCGS would be to allow more exploration time per frames. The MCGS algorithm is highly parallel as it can explore different nodes and different paths at the same time. By parallelizing its algorithm, exploration time could be multiplied providing better results or faster adaptation.

Finally, Task Skipping may be improved by using a simplified version of the tasks instead of simply delaying it. This would result in a smoother degradation of the quality of execution while providing enhanced performance. However this suppose implementing several version of skippable tasks.

The hardware developments of the different platforms on which video games can be executed have been and remain driving forces in the evolution of computing power. The arrival of TPUs, ARM architectures, and disaggregated architectures (Meteor Lake, Arrow Lake **Fig.5.1**) suggest a paradigm shift in the operation of both consoles and players' PCs in the medium to long term. In this context, it is necessary to have a dynamic scheduling of heterogeneous tasks in order to direct the computations not only based on their nature but also based on the execution, allowing greater execution versatility and better use of all the components of a machine.

Finally, the study and optimization of energy consumption in computer applications,

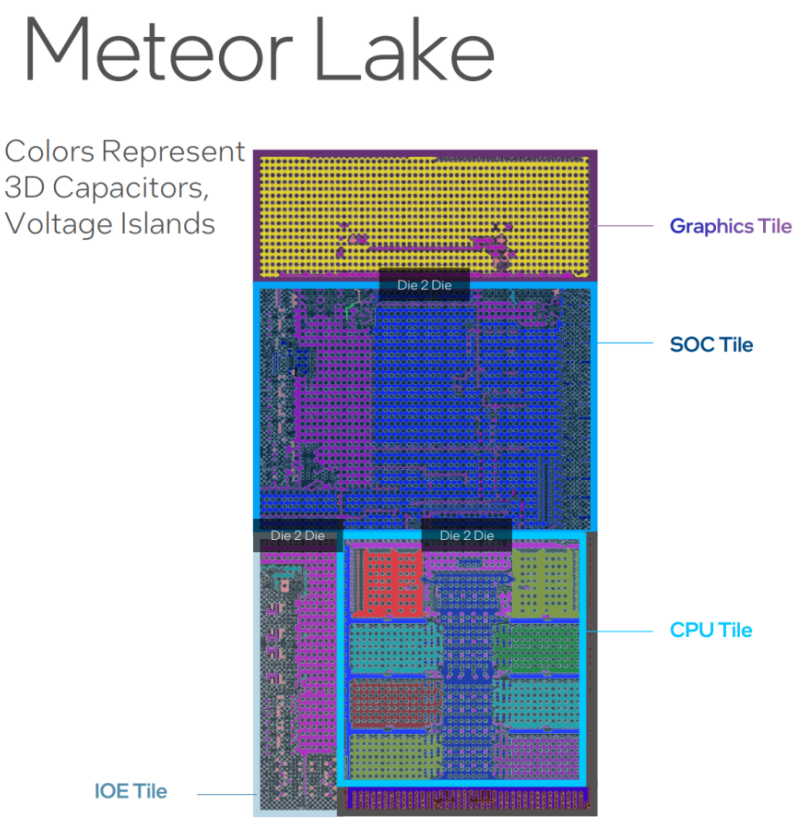



Figure 5.1: **Example of disaggregated architectures with the MeteorLake from Intel**

particularly video games, represent a major current issue, whether it is to extend the autonomy of portable devices (Switch/SteamDeck), offer players the possibility to manage their energy consumption, or reduce electricity consumption and associated costs in a crisis context.In this context, simply reducing the game's quality (graphics or framerate) does not guarantee a significant or effective reduction in the machine's energy consumption. Therefore, we propose to study the possibilities of optimization and adaptation of game engines in a limited energy resource environment.

# Résumé

Cette thèse de doctorat se déroule dans le cadre d'une convention CIFRE entre l'équipe STORM de l'Inria Bordeaux et Ubisoft.

Dans le processus de développement d'un jeu vidéo, les équipes se heurtent à un problème majeur: les performances du jeu sur console et PC. En effet, les jeux AAA sont gourmands en puissance de calcul et ceux-ci doivent être capable de garantir la construction d'au moins 30 images par seconde sur consoles et 60 images par seconde sur PC. Afin de construire une image (frame), un ensemble de tâches doit être exécuté sur la machine d'un joueur, comme le moteur physique, la gestion du son, les intelligences artificielles. Chacune de ces tâches peut être exprimée de façon monolithique ou décomposée en un petit graphe parallèle. Elles sont regroupées sous forme d'un graphe global et doivent être exécutées efficacement afin de garantir un certain taux de rafraîchissement. Par ailleurs il est nécessaire de respecter les dépendances entre les tâches et leurs priorités afin de conserver à la fois une cohérence lors de l'exécution du jeu, mais également utiliser au mieux le temps disponible: cela veut dire qu'il faut utiliser efficacement la mémoire et le parallélisme de la machine.

Le problème de l'ordonnancement est bien connu des chercheurs mais le cas du jeu vidéo est très particulier. En effet un moteur de jeu peut être considéré comme une système 3D interactif en temps réel "souple", car si la limite temporelle que représente la construction de la frame n'est pas respectée, le jeu ne s'arrête pas et nous n'avons pas de risques de bugs majeurs contrairement aux exécutions temps réelles "dures" mais seulement une perte significative de confort pour le joueur, résultant en une exécution saccadée ainsi que des arrêts sur images le temps que le moteur puisse finaliser l'ensemble des calculs demandés.

Une première difficulté réside dans le fait que ces tâches sont construites et ajoutées par des personnes différentes, dont les métiers sont indépendants. Le graphe de tâches global est obtenu par composition et est très difficilement accessible lors de la conception du jeu vidéo. Par ailleurs, le nombre de tâches et leur durée d'exécution varie au cours du déroulement du jeu en fonction de la position du joueur, de ses actions, de son environnement ou des autres joueurs. De plus la nécessité de respecter les contraintes de temps réel peuvent en fonction des ressources de calculs imposer l'utilisation de versions dégradées de certaines tâches voir complètement les abandonner. De plus les méthodes normalement utilisé pour ordonnancer les système temps réels mou ne peuvent être appliqué dans notre cas.

Dans cette thèse, nous analysons dans un premier temps l'impact de méthodes issue du calcul haute performance sur les moteurs de jeux et proposons une formalisation du problème de l'ordonnancement temps réel souple dans le cadre du jeu vidéo. Nous présentons également un simulateur permettant de générer des frames représentatives du problème implémenté à partir de traces d'exécution collectées lors de parties. Ce simulateur nous permet d'évaluer les opportunités d'optimisation sans avoir à implémenter ces changements directement dans le moteur de jeu.

De cette façon, nous montrons les gains potentiels pouvant être obtenus en mettant en place des heuristiques qu'elles soient dynamiques ou statiques, nous présentons et évaluons les performances de 12 différentes méthodes dans ce contexte particulier qu'est la simulation interactive en 3D. Nous présentons également les gains envisageable via la mise en place d'un ordonnancement à plusieurs niveau permettant de gérer à la fois les "Macro

tâches" définies directement dans le graphe de tâche ainsi que les "Micro tâches" générées à la volée lors de l'exécution dont nous ne pouvons connaître le nombre ni la longueur à l'avance. Finalement, nous investiguons les optimisations envisageable au travers de la parallélisation de tâches ciblées du moteur. De cette analyse résulte la nécessité d'avoir un ordonnancement dynamique de tâches. L'un des intêrét de ce choix est de prendre en compte les variations des différentes tâches induisant par la même des changements de chemin critique dans le graphe d'exécution. Différentes heuristiques d'ordonnancement dynamique existent pour réaliser cela, cependant elles ont une complexité trop élevée pour être recalculées à chaque frame et ainsi garantir une adaptabilité satisfaisante. Une solution pour palier à ce coup de calcul élevé sur une frame donnée est d'améliorer de façon incrémental l'ordonnancement sur un grand nombre d'itération afin d'obtenir un ordonnancement proche de l'optimal pour un moment donner de l'exécution et d'adapter de façon dynamique cette dernière tout au long de la session de jeu tout en bornant le temps de calcul alloué à ces optimisations.

Le contexte du jeu vidéo nous apporte des informations et des possibilités jusque-là non présentes dans les problèmes d'ordonnancements classiques, en effet la connaissance en détail de la frame précédente, la relative continuité de la charge de calcul à réaliser par les processeurs lors d'une phase de jeu ainsi que la redondance du graphe de tâche à exécuter a très haute fréquence nous fournis tous les éléments nécessaires pour mettre en place une méthode incrémentale de détermination de l'ordonnancement.

Nous proposons ainsi dans un deuxième temps, une méthode d'ordonnancement dynamique, basées sur l'exploration d'un graphe de Monte Carlo afin de déterminer les optimisations locales possibles et ainsi adapter l'exécution de la Frame en fonction des besoins du jeu en priorisant les parties demandant le plus de ressources à un instant T de façon dynamique et en temps et espace contrôlés.

Les méthodes de Monte Carlo sont une famille d'algorithme visant à calculer une valeur numérique approchée en utilisant des tirages aléatoires, publié pour la première fois en 1949 dans un article de Nicholas Metropolis. Elles ont connu un premier essor sous l'impulsion de John von Neumann lors de la seconde guerre mondiale pour résoudre les équations aux dérivées partielles dans le cadre du “Monte-Carlo N-Particle transport”. A partir des années 2010 un deuxième élan pour cette famille d'algorithme apparaît grâce au mathématicien Rémi Coulom qui l'applique au jeu de Go sous la forme d'un arbre de recherche des coups possibles, c'est sur cette méthode que sont basé les programmes AlphaGo et AlphaZero qui triomphent des champions mondiaux de Go, considéré jusque-là impossible dû à la composante combinatoire du problème.

Nous montrons au travers de plusieurs expérience que l'utilisation du MCGS dans le cadre d'un ordonnancement dynamique de tâche permet à l'execution d'être plus robuste et de mieux résister aux fortes variations de charges que nous pouvons observer dans le jeu vidéo. Cependant, un certains manque de gains en performance venant directement du manque de parallélisme des moteurs de jeu est apparent. Si un travail de fond sur l'optimisation et la parallélisation de ces dernier est effectué au cours des prochaines années, nous sommes confiant sur le fait que cette méthode puisse apporter des gains substanciels.

Une autre façon de garantir un meilleur ordonnancement des frames et d'empêcher une baisse du taux de rafraichissement de l'image est de jouer directement sur l'exécution en elle-même. Dans le cadre d'un moteur de jeu, certaines tâches peuvent être considérées comme “Skippeable” c'est à dire qu'elles peuvent être reportées sur plusieurs frame

jusqu'au moment ou ces dernières deviennent critiques et doivent alors être exécutées pour ne pas trop dégrader l'expérience du joueur. Parmi ces dernières nous pouvons bien évidemment citer les chargements, la gestion du Path-Finding ou des Personnages Non-Joueurs (NPC) ou encore les explosions.

Lors de l'exploration du Monte Carlo Graph Search (MCGS) dans le but d'améliorer itérativement l'ordonancement, la phase de simulation fournit de nombreuses informations sur le comportement attendu pour la prochaine image.

Nous proposons ainsi grâce au MCGS d'anticiper si l'image suivante ne respectera pas le seuil prédéterminé et de déterminer la quantité de ces tâches que nous pouvons exécuter lors d'une frame donnée. De cette façon nous pouvons adapter dynamiquement l'execution aux ressources disponibles sur la machine du joueur plutôt que l'inverse.

Nous avons évalué cette méthode sur plusieurs exécution de jeux. Les résultats obtenus montrent une plus grande tolérances aux variations de charge dans le moteur. De plus, lors par exemple de chargements ou encore lorsque le système d'exploitation se voit dans l'obligation d'effectuer des tâches externes à l'execution du jeu générant une perturbation de l'exécution par la préemption d'un cœur de calcul sur les processeurs à disposition. Nous avons montrons que ce mécanisme, gomme les perturbations et empêche le ralentissement du jeu.

# Bibliography


[1] Example of game flickering in satisfactory. https://www.youtube.com/watch?v=oCCQunhSXPI&ab_channel=n.steam, 2019.

[2] Unreal Engine 5 Documentation. https://docs.unrealengine.com/5.0/en-US/tasks-systems-in-unreal-engine/, 2024. Accessed: 2022-01-26.

[3] 3dfx. Programming the 3dfx interactive glide™ rasterization library 2.2. https://www.gamers.org/dEngine/xf3D/glide/glidepgm.htm, 1997.

[4] Adam, T. L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. *Commun. ACM 17*, 12 (Dec. 1974), 685–690.

[5] Alfieri, B. Intel games task scheduler. https://github.com/GameTechDev/GTS-GamesTaskScheduler, Jan. 2019. Accessed: 2022-01-04.

[6] Asta, S., Karapetyan, D., Kheiri, A., Özcan, E., and Parkes, A. J. Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences 373* (2016), 476–498.

[7] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2011), 187–198.

[8] Beaumont, O., Eyraud-Dubois, L., and Gao, Y. Influence of tasks duration variability on task-based runtime schedulers. *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), 16–25.

[9] Benoit, A., Canon, L.-C., Elghazi, R., and Héam, P.-C. Update on the Asymptotic Optimality of LPT. In *Euro-Par 2021: Parallel Processing* (Cham, 2021), L. Sousa, N. Roma, and P. Tomás, Eds., Springer International Publishing, pp. 55–69.

[10] Bruno, J., Coffman, E. G., and Sethi, R. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM 17*, 7 (July 1974), 382–387.

[11] Caccamo, M., and Buttazzo, G. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *IEEE Real-Time Systems Symposium* (San Francisco, Dec 1997), IEEE, pp. 330—339.

[12] Cai, X., Wu, X., and Zhou, X. *Optimal stochastic scheduling*, vol. 5. Springer, 2014.

[13] Canon, L.-C., and Jeannot, E. Evaluation and Optimization of the Robustness of DAG Schedules in Heterogeneous Environments. Research report, 2008.

[14] Chakra, A. A., and Li, X. Priority-based level of detail approach for animation interpolation of articulated objects.

[15] Coffman, E. G., and Graham, R. L. Optimal scheduling for two-processor systems. *Acta informatica 1*, 3 (1972), 200–213.

[16] Cournoyer, F. Massive crowns on Assassin's Creed Unity: AI Recycling. GDC 2015 `https://www.youtube.com/watch?v=Rz2cNWVLncI&ab_channel=GDC`, 2015.

[17] Czech, J., Korus, P., and Kersting, K. Improving alphazero using monte-carlo graph search. *Proceedings of the International Conference on Automated Planning and Scheduling 31*, 1 (May 2021), 103–111.

[18] Dai, G., Mohaqeqi, M., and Yi, W. Timing-anomaly free dynamic scheduling of periodic dag tasks with non-preemptive nodes. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (2021), pp. 119–128.

[19] Graham, R., Lawler, E., Lenstra, J., and Kan, A. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Discrete Optimization II*, vol. 5 of *Annals of Discrete Mathematics*. Elsevier, 1979, pp. 287–326.

[20] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (1969), 416–429.

[21] GrandViewResearch. Video Game Market Size, Share and Trends Analysis Report By Device (Console, Mobile, Computer), By Type (Online, Offline), By Region (North America, Europe, Asia Pacific, Latin America, MEA), And Segment Forecasts, 2022 - 2030. `https://www.grandviewresearch.com/industry-analysis/video-game-market`. Accessed: 2023-01-25.

[22] Gregory, J. *Game engine architecture*, 3 ed. Taylor and Francis Ltd., New York, 2018.

[23] Horn, W. A. Technical note—minimizing average flow time with parallel machines. *Operations Research 21*, 3 (1973), 846–847.

[24] Hu, T. C. Parallel sequencing and assembly line problems. *Operations Research 9*, 6 (1961), 841–848.

[25] IPSOS. Essential fact about the video game industry. `https://www.theesa.com/wp-content/uploads/2021/08/2021-Essential-Facts-About-the-Video-Game-Industry-1.pdf`, 2021.

[26] Kocsis, L., and Szepesvári, C. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006* (Berlin, Heidelberg, 2006), J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Springer Berlin Heidelberg, pp. 282–293.

[27] Koren, G., and Shasha, D. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings 16th IEEE Real-Time Systems Symposium* (1995), pp. 110–117.

[28] Leung, J. Y. *Handbook of scheduling: algorithms, models, and performance analysis.* CRC press, 2004.

[29] Leurent, E., and Maillard, O.-A. Monte-carlo graph search: the value of merging similar states. In *Proceedings of The 12th Asian Conference on Machine Learning* (18–20 Nov 2020), S. J. Pan and M. Sugiyama, Eds., vol. 129 of *Proceedings of Machine Learning Research*, PMLR, pp. 577–592.

[30] Limpert, E., Stahel, W. A., and Abbt, M. Log-normal Distributions across the Sciences: Keys and Clues: On the charms of statistics, and how mechanical models resembling gambling machines offer a link to a handy way to characterize log-normal distributions, which can provide deeper insight into variability and probability—normal or log-normal: That is the question. *BioScience 51*, 5 (05 2001), 341–352.

[31] Marchal, L., Simon, B., Sinnen, O., and Vivien, F. Malleable task-graph scheduling with a practical speed-up model. *IEEE Transactions on Parallel and Distributed Systems 29*, 6 (2018), 1357–1370.

[32] Nascimento, F. M. S., and Lima, G. Effectively scheduling hard and soft real-time tasks on multiprocessors. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2021), pp. 210–222.

[33] Nascimento, F. M. S., and Lima, G. Effectively scheduling hard and soft real-time tasks on multiprocessors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2021), pp. 210–222.

[34] Regragui, M., Coye, B., Pilla, L. L., Namyst, R., and Barthou, D. Exploring scheduling algorithms for parallel task graphs : a modern game engine case study. In *Euro-Par Conference* (Glasgow, 2022, Aug. 2022).

[35] Saffidine, A., Cazenave, T., and Méhat, J. Ucd : Upper confidence bound for rooted directed acyclic graphs. *Knowledge-Based Systems 34* (2012), 26–33. A Special Issue on Artificial Intelligence in Computer Games: AICG.

[36] Schaller, R. Moore's law: past, present and future. *IEEE Spectrum 34*, 6 (1997), 52–59.

[37] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T. P., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nat. 550*, 7676 (2017), 354–359.

[38] Steam. Steam game release summary. `https://steamdb.info/stats/releases/`, 2023.

[39] Unity User Manual 2020.3 (LTS). `https://docs.unity3d.com/Manual/UnityManual.html`, 2023. Accessed: 2022-01-26.

[40] Unreal Engine 4 Documentation. `https://docs.unrealengine.com/4.27/en-US/`, 2023. Accessed: 2022-01-26.

[41] UnrealEngine. Unreal engine 4 and 5 technical demo. https://www.youtube.com/watch?v=vV487suNJOA&ab_channel=GamerInVoid, 2020.

[42] Vreman, N., Pates, R., and Maggio, M. Weaklyhard.jl: Scalable analysis of weakly-hard constraints. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2022), pp. 228–240.

[43] Wałędzik, K., and Mańdziuk, J. Applying hybrid monte carlo tree search methods to risk-aware project scheduling problem. *Information Sciences 460-461* (2018), 450–468.

[44] wolfpld. Tracy profiler. https://github.com/wolfpld/tracy, 2022.

[45] Yakal, K. The evolution of commodore graphics. Compute!'s Gazette. pp. 34–42. Retrieved 2019-06-18., June 1986.

[46] Yuki, N. Peak video game? top analyst sees industry slumping in 2019. Bloomberg L.P. Archived, January 23, 2019.

[47] Świechowski, M., Godlewski, K., and Sawicki, B. e. a. Monte carlo tree search: a review of recent modifications and applications. *Artif Intell Rev* (July 2022).