Exploring scheduling algorithms for parallel task graphs: a modern game engine case study

Mustapha Regragui, Baptiste Coye, Laércio Lima Pilla, Raymond Namyst, Denis Barthou

Exploring scheduling algorithms for parallel task graphs: a modern game engine case study

Mustapha Regragui[1], Baptiste Coye[1,2], Laércio L. Pilla[1], Raymond Namyst[1], and Denis Barthou[1]

[1]Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800, Talence, France
[2]Ubisoft Bordeaux
email: mustapha.regragui@bordeaux-inp.fr, baptiste.coye@ubisoft.com, {laercio.lima-pilla, raymond.namyst, denis.barthou}@inria.fr

Abstract

Game engines are at the heart of the design of modern video games. One of their functions is to keep a high frame rate by scheduling the tasks required to generate each frame (image). These tasks are organized in a soft real-time, parallel task graph, which is a scenario very few works have focused on, or adapted scheduling algorithms to. In this paper, we study the scheduling problem of game engines. We model the tasks and the scheduling problem by profiling a commercial game engine, adapt and compare different scheduling algorithms, and propose two additional scheduling optimizations.

Keywords: Scheduling; Video Game Engine; Soft Real-Time.

1 Introduction

The video game market is valued at over 100 billion USD [14], making it larger than the HPC market, or even the movie and music industries combined. This industry impacts computing both at the hardware and software levels. It produces and sells tens of millions of video game consoles yearly. Its games are run on personal computers, consoles, smartphones, and even on Cloud gaming servers. These games are more easily developed and ported to different platforms thanks to a key software component called the game engine.

The **game engine** serves as a framework for game development. Video game companies can license game engines, such as Unity, Unreal Engine 4, or Game Maker: Studio, or produce their own. A game engine contains core software components (such as the 3D rendering system or the collision detection system) that can be extended and combined with different art assets and game logic to produce different games [11]. Moreover, game engines are responsible for managing resources such as the memory, and for scheduling tasks. This makes the optimization of game engines essential for the gaming experience of players.

The problem of scheduling in game engines has not been previously studied in detail, and we have noticed that this problem includes uncommon characteristics. For instance, although video games are soft real-time systems (with a video frame being displayed every few milliseconds), their recurring tasks do not match exactly with the models of periodic or sporadic task systems [15, Chapter 28]. Additionally, although their tasks are parallel, they are neither malleable nor moldable [15, Chapter 25]. With a better understanding of this problem, we could optimize the schedule in game engines for the benefit of players and developers alike: with fewer dropped frames, players can have a better gaming experience; with some extra free time before having to display a frame, developers can include more detailed graphics, physics, or AI, while mobile devices can operate at lower frequencies and save battery, and Cloud gaming servers can dedicate less computing resources to a game.

In this paper, our goal is to understand and find solutions to the scheduling problem of game engines. Working with a modern game engine as a case study, we are able to model this problem and to adapt several scheduling algorithms to it. Our experimental evaluation reveals performance improvements in the game engine when answering the following three research questions: (I) "*Can scheduling strategies from the state of the art improve the performance of game engines?*"; (II) "*Can changes in the scheduling mechanism of a game engine reduce the performance gap between schedulers and the critical path?*"; and (III) "*Can small changes to the task graph lead to performance improvements?*". Our contributions include the model and evaluation metrics, the list of algorithms adapted for this problem, and their experimental evaluation, which are all presented in Sects. 3, 4, and 5, respectively. Sect. 2 covers information on game engines and related work, and Sect. 6 provides concluding remarks.

2 Background and Related Work

2.1 Background on video games and game engines

Video games work as soft real-time interactive simulations [11]. The frequency of interactions is given by the *frame rate*, which defines how many *frames* (images) are presented *per second*. Nowadays, the de facto standard dictates a frame rate of 60fps, giving $\frac{1}{60}s \approx 16.667ms$ for producing each frame. If this time is surpassed, a frame is *dropped*. Frequent frame drops degrade the gaming experience.

As a video game is built on a game engine, engineers will add, remove, or adapt functionalities to their needs. Given the complexity of developing a game engine, it is very common for large companies to maintain their own engines for over a decade to capitalize on their know-how. Still, the cost of hand-tuning all their features is exacerbated by the variety of gaming platforms and their evolution (e.g., increase in the number of CPU cores available).

One way of reducing the required hand-tuning comes with the game engine scheduler, which manages the execution of tasks (functionalities) by itself. For instance, Unity provides its Job System [18] so engineers can write their own tasks with dependencies and let Unity schedule them. Meanwhile, Unreal Engine 4 includes Tick Groups [19] to set when a task should be executed (e.g., before or after physics simulations). Our case study contains a task graph that

is executed for each frame. The structure of the graph is static for a single game, and it changes very little between games, making it representative. This organization makes it easier for multiple teams of engineers to work on their own functionalities. Additionally, each task is composed of one or more parallel subtasks, which is also recommended on Intel's Games Task Scheduler (GTS) [3].

2.2 Scheduling problems and algorithms

The game engine's scheduler may find issues to keep the frame rate due to resources being used by others (e.g., the operating system), for other tasks (loading assets), or due to changes in the **load of the game** (e.g., additional objects to render or AI agents to simulate). A change in load may mean not only a change in the execution time of a task, but also to its number of subtasks. Although high load episodes may be hard to anticipate (given the dynamic and interactive nature of video games), load changes are usually gradual. A scheduler could benefit from this by estimating the behavior of tasks based on recent frames in a way similar to the use of the principle-of-persistence in periodic load balancing [1]. Conversely, estimations are avoided by GTS through work stealing [3].

Given the lack of studies on this scheduling problem, our efforts have been dedicated to finding and adapting algorithms and heuristics proposed in other contexts [2,7,9,12,13] (cf. Sect. 4). We find that there is value in bringing to light new applications and knowledge on existing algorithms, as have done Benoit et al. [5] for the asymptotic performance of the longest processing time (LPT) heuristic for the case of tasks originating from uniform integer compositions.

Our scheduling problem has distinct characteristics that block the use of techniques and heuristics used for scheduling traditional real-time or parallel tasks [15]. Real-time scheduling most often considers independent, recurring tasks. Such is the case on the work of Nascimento and Lima [16], where earliest deadline first (EDF) heuristics are employed for scheduling soft and hard real-time tasks in parallel resources. Nonetheless, the game engine contains dependent tasks with an entire task graph to be computed for each frame. Additionally, all tasks share the same due date, obstructing the use of EDF heuristics. Meanwhile, parallel task scheduling usually models tasks that use multiple resources simultaneously, but the game engine's tasks follow a fork-join model internally. These levels of tasks and subtasks are also reflected on GTS [3] with its **macro-** and **micro-schedulers**. An algorithm called DynFed was proposed to schedule parallel tasks with dependencies in real-time systems by Dai, Mohaqeqi, and Yi [8]. Nonetheless, it focuses on periodic, independent tasks whose parallel subtasks have dependencies, while our scheduling problem contains tasks with dependencies whose parallel subtasks are independent.

3 Scheduling in Game Engines

Our discussion of the scheduling problem in game engines is organized in three parts: the task model; the scheduling problem at the scale of a single frame; and the problem for multiple frames.

3.1 Task model

A game engine performs multiple tasks to produce each frame (e.g., graphics rendering and physics simulations [11]). These tasks have precedence constraints that must be respected for their correct execution, which leads to their organization as a directed acyclic graph (DAG). Fig. 1 represents the task graph of our case study. It was extracted from a modern video game from Ubisoft and its structure is reflected in other game engines and video games. The leftmost task is the start of the frame and the rightmost its end. The path on the bottom of Fig. 1 is composed of graphic tasks (all run in the same CPU core to dispatch work to the GPU), while the other paths represent simulation and control tasks.

Each task represents a functionality written by a given team in a given moment in the lifetime of the game engine, so task interactions have to be kept simple. Internally, each task contains one or more independent, sequential subtasks following a fork-join model. For our ≈ 100 tasks, over 1000 subtasks can be computed at each frame. Both their number and execution time may change during the game execution. We refer to this effect as the *load* of the frame.

In order to model and simulate the behavior of the game engine under different loads, we profiled its tasks and subtasks on varied executions (> 10 on different maps of the game) and different phases (over 3000 frames). We obtained their minimum, maximum, mean, and standard deviation values, and used them to model timings as log-normal distributions [17] depending on the load. Eq. 1 defines the processing time p_j^{sub} of a subtask of task j with load $l \in [0, 1]$ depending on $p_j^{min}(l)$, $p_j^{max}(l)$, $\mu_j(l)$, and $\sigma_j(l)$ that are resp. the minimal, maximal, mean, and standard deviation of the execution time under load l. Each value was obtained for *low* ($l = 0$) and *high* ($l = 1$) loads, and intermediary values are computed by a linear interpolation in l. We compute the processing time $p_j(l)$ of task j by adding together the times of its subtasks in Eq. 2. In it, $s_j(l)$ represents the number of subtasks of task j with load l, which is computed in a similar fashion to other load-dependent parameters — i.e, $s_j(l) = \lceil (1 - l) \cdot s_{j,low} + l \cdot s_{j,high} \rceil$.

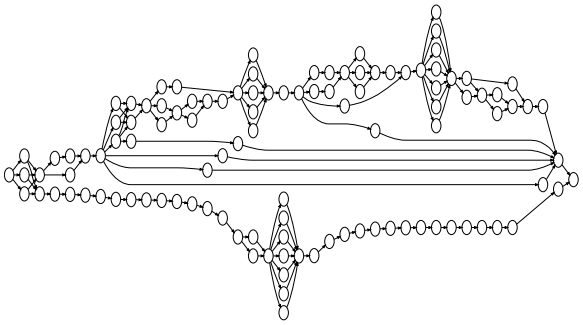


Figure 1: A DAG representing game engine tasks (vertices) and their precedence constraints (edges, from left to right).

$$p_j^{sub}(l) = p_j^{min}(l) + (p_j^{max}(l) - p_j^{min}(l)) \log \mathcal{N}(\mu_j(l), \sigma_j(l)) \quad (1)$$

$$p_j(l) = \sum_{k=1}^{s_j(l)} p_j^{sub}(l) \quad (2)$$

3.2 Scheduling problem for a single frame

A simplified description of the scheduling problem for a frame can be shown using Graham's notation [10]. The machine environment is composed of parallel and identical resources (CPU cores). The task characteristics and scheduling constraints follow the model of Sect.3.1. In short, our tasks have precedence constraints, different processing times, and the same due date. For the objective function, we use the tardiness T_j to evaluate if the due date was respected.

Given the aforementioned characteristics, this scheduling problem can be represented as $P|prec, d_j = d|T_{max}$, which is NP-Hard. Still, this does not capture all the details of our problem in practice, mainly due to imprecision on the processing times of tasks. Our tasks are modeled using stochastic processing times, and time-ware scheduling algorithms are mostly dependent on measurements from previous frames to estimate the current frame's behavior (i.e., its load). This is not an issue thanks to the stability of the game engine and to the minor effects of slight prediction disturbances in similar contexts [4]. In this sense, using the notation P_j to represent stochastic processing times [6, Chapter 1], our scheduling problem would be closer to $P|P_j, prec, d_j = d|T_{max}$.

3.3 Scheduling problem for multiple frames

The quality of a scheduling solution for multiple frames is based on its results for each frame. 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_{max}^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), represented in Eqs. 3, 4, and 5. 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.

$$SF = \max_{f \in [1,F]} T_{max}^f \quad (3)$$

$$DF = \sum_{f \in [1,F] \wedge T_{max}^f > 0} 1 \quad (4)$$

$$CS = \sum_{f \in [1,F] \wedge T_{max}^f > 0} T_{max}^f - d \quad (5)$$

4 Exploring List Scheduling Algorithms

Given the absence of known solutions for our scheduling problem, we have selected — and, sometimes, adapted — several scheduling algorithms used in other contexts to our experiments. All of them follow a list scheduling strategy: Whenever a resource becomes available, the macro-scheduler takes the task with

the highest priority and the micro-scheduler executes one of its subtasks. Besides its known benefits, list scheduling is also attractive for its ability to adapt to changes in the number of resources available.

Table 1 lists the chosen algorithms, which we believe cover a wide range of the behaviors seen in the literature. The algorithms are ordered according to the way they compute task priorities. *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.

The First In, First Out scheduler represents the original implementation in the game engine and serves as the baseline. Regarding online algorithms, SLPT (and SSPT) follows the same logic of LPT [9] (SPT [12]), but at a subtask level (i.e., using $p^{sub}_{j,k}$). Instead of using 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 (r_j), its first subtask starts executing (b_j), and its last subtask finishes executing (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 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.

Offline algorithms try to prioritize tasks that may delay the completion of the last task (*exit node*). HLF [13] prioritizes tasks in the longest paths to the exit node, while HLFET [2] extends it with processing time estimations (the mean times used in our model). CG [7] uses a labeling algorithm that has been shown to be optimal for the problem $P2|p_j = p, prec|C_{max}$.

Our last algorithm, named DCP, combines global information online. 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. 6 using information from the previous frame and the set of successors of task j in the graph as $succ(j)$.

$$prio(j) = \max_{i \in succ(j)} prio(i) + \max\left(\frac{\sum_{k=1}^{s_j} p^{sub}_{j,k}}{\min(m, s_j)}, \max_{k \in [1,s_j]} p^{sub}_{j,k}\right) \tag{6}$$

Table 1: Characterization of tested scheduling algorithms.

Acronym	Ref.	Meaning	Info. scale	Priority comp.	Time-awareness
FIFO		First In, First Out	—	—	—
LPT	[9]	Longest Processing Time first	Local	Online	Previous frame
SPT	[12]	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	Prev. & curr. frame
WT		Longest Waiting Time first	Local	Online	Prev. & curr. frame
HLF	[13]	Highest Level First	Global	Offline	—
HLFET	[2]	HLF with Estimated Times	Global	Offline	Mean
CG	[7]	Coffman-Graham's algorithm	Global	Offline	—
DCP		Dynamic Critical Path	Global	Online	Previous frame

5 Experimental Evaluation

We conducted a series of experiments using an in-house simulator covering three different scenarios based on the research questions brought up in Sect. 1. Following the methods described in Sect.5.1, the results of the experimental scenarios are presented in Sects. 5.2, 5.3, and 5.4.

5.1 Details regarding the simulation and statistical evaluation

The experiments use an in-house scheduling simulator written in C++. Given a complete description of the task graph (Sect. 3.1), 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. This enables direct comparisons between scheduling algorithms and experimental scenarios. The simulation represents an ideal environment with no overhead from the scheduling algorithm, data locality, or other sources of interference, trading realism for understandability. All parameters required to model the tasks (Eq. 1) were obtained in a development machine from Ubisoft.

To test load variations, each simulation runs 200 frames with the load parameter starting at 0 and increasing linearly up to 1 in the 101^{st} frame and then decreasing linearly until it reaches 0.01 for the last frame. This provides a gradual change of load while also generating a load peak. For each scenario and scheduling strategy, we ran simulations using from 4 up to 20 resources. By regarding results with fewer resources, we can also anticipate the effects of external interference (Sect. 2.2). Our standard case is set to 12 resources, as this is a common number of cores in current gaming processors. In each situation, we varied the RNG seed in the interval $[1, 50]$. Excluding Critical Path simulations, this represents a total of $200 \times 50 \times 17 \times 11 \times 3 = 5,610,000$ frames.

For the statistical evaluation of our experiments, we first employed descriptive methods to understand our results and to verify that no errors were present. We then followed with inferential methods. Setting our tests to a 5% significance level, we used Kolmogorov-Smirnov tests to check if samples came from normal distributions for all metrics 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 results were obtained on an Intel Core i7-1185G7 processor, with 32GB of LPDDR4 RAM (3200MHz). The machine ran on Ubuntu 20.04.3 LTS (5.13.0-1022-oem), and g++ 9.3.0 was used for the simulator's compilation (-O3 flag).

5.2 Scenario I - employing scheduling algorithms

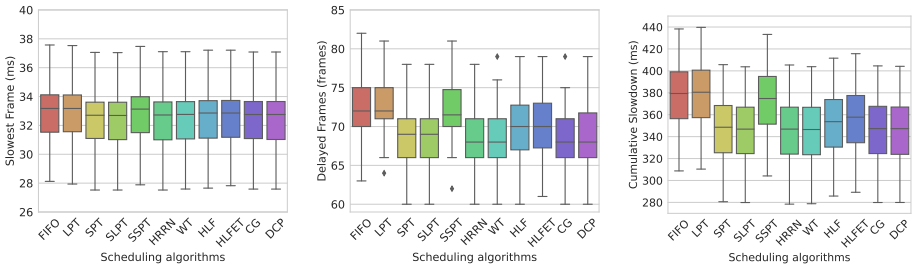
We summarize the main performance results when scheduling tasks on 12 resources in Table 2 and Fig. 2 (small values the better). Table 2 shows the values of Slowest Frame, Delayed Frames, and Cumulative Slowdown (rows) computed for each scheduling algorithm (columns). These values represent the averages over 50 executions. The first column presents FIFO (our baseline) and the last

column shows the values for the Critical Path. The general distribution of values for the different metrics is illustrated as boxplots in Fig. 2.

The smallest improvements are achieved for the SF metric. This indicates that, under the worst load conditions, no algorithm is able to avoid the large increase in frame duration. Still, even the minor improvements achieved by WT and CG are still statistically significant (p-values = 5.12×10^{-33} and 9.64×10^{-30}, resp.). This is not the case for LPT (p-value = 0.69). In any case, the average SF for the Critical Path is only better than FIFO's by a factor of 1.159, and still 1.70 times larger than the desired frame duration (16.667 ms).

Table 2: Average metrics for all scheduling strategies on 12 resources.

	FIFO	LPT	SPT	SLPT	SSPT	HRRN	WT	HLF	HLFET	CG	DCP	Crit. 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



(a) Slowest Frame (b) Delayed Frames (c) Cumulative Slowdown

Figure 2: Boxplots for the 3 metrics on 12 resources. Vertical axes start at different points to emphasize differences.

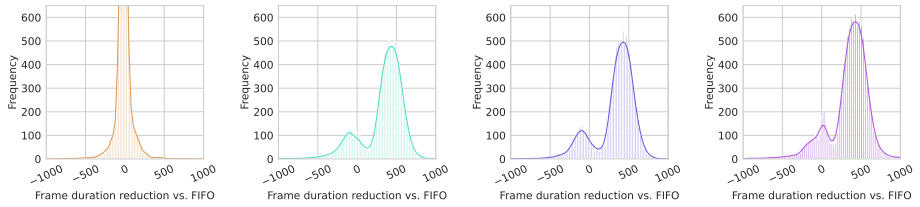
The scheduling algorithms provide more noticeable improvements for the DF and CS metrics. This happens for strategies both local and global, online and offline (Table 1). For instance, WT (local, online) reduced DF by a factor of 1.054 over the baseline (p-value = 4.15×10^{-21}), as did CG (global, offline) (p-value = 5.65×10^{-21}). DCP (global, online) did the same by a factor of 1.055 (p-value = 4.21×10^{-21}). Interestingly enough, we cannot say that these three strategies perform differently for the DF metric (all p-values > 0.05), but we can do so for CS (all p-values < 0.05).

In order to better understand the effects of the scheduling algorithms on the duration of the frames, Fig. 3 shows the frame duration reductions achieved by LPT, WT, CG, and DCP as histograms. These values are obtained 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]. A positive reduction means that the algorithm reduces the duration of a specific frame, thus improving performance.

Three relevant aspects can be noticed here. First, LPT (Fig. 3a) has most of its frame duration reductions centered around $0\mu s$, indicating that its decisions

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

lead to schedules very similar to FIFO. Second, the other illustrated strategies have results mostly centered around $500\mu s$ with slightly different curves. Although they make different 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.



(a) LPT (b) WT (c) CG (d) DCP

Figure 3: Histograms presenting frame duration reductions (in μs) compared to FIFO (positive values mean shorter frame durations by the algorithms). Lines represent kernel density estimations.

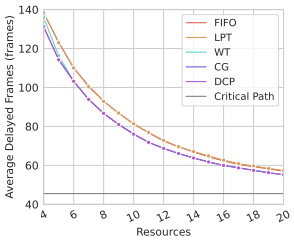


Figure 4: Average number of delayed frames on different number of resources. The vertical axis starts at 40 frames to emphasize differences.

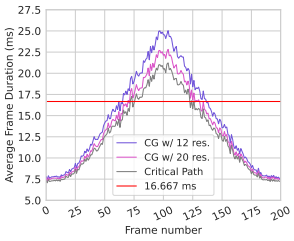


Figure 5: Average duration of each frame for CG on 12 and 20 resources, and for the Critical Path.

Performance improvements can also be seen across different numbers of resources. This is illustrated in Fig. 4, where the average DF of selected algorithms (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. The absolute difference between FIFO and other strategies tends to decrease when more resources are available, going from about 7 frames on 4 resources down to under 2 frames on 20 resources. This shows that it becomes harder to saturate resources as their numbers increase, which in turn reduces the delay seen on important tasks from the critical path.

Even when scheduling tasks on 20 resources, a noticeable gap remains between some of the best schedulers and the Critical Path. To better illustrate this difference, Fig. 5 contrasts the frame duration of the Critical Path and CG. The horizontal axis represents the simulated frames in order, while the vertical axis represents their average durations for CG with 12 and 20 resources, and for the Critical Path. Fig. 5 exposes the change in frame duration following the

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

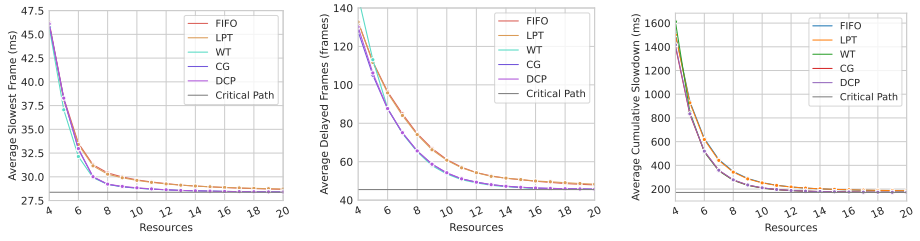
5.3 Scenario II - subtask scheduling

In search of a way to overcome the previous limitations, we have moved our attention from the macro-scheduler to the micro-scheduler (cf. Sect. 4). Originally, the micro-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. We consider this is a feasible change to the game engine because it does not affect the actual execution of the subtasks nor the dependencies in the task graph. Additionally, developers can provide clues of the most important subtasks statically or using simple internal parameters.

The performance results achieved in this scenario are summarized in Table 3. Its additional rows show how much the metrics have been reduced in comparison to Scenario I (Table 2). The improvements are noticeable for all scheduling algorithms and metrics. For instance, the average SF for FIFO changed from $32.88ms$ to $29.29ms$, representing a 10.93% decrease in time (an improvement factor of 1.123). This is greater than the benefits previously achieved by changing the scheduling algorithms only. Still, in many cases, the algorithms show even better gains, leading to greater cumulative improvements over FIFO.

Table 3: Average metrics for all schedulers over 12 resources with sorted subtasks. Percentage reductions are calculated in comparison to Table 2.

	FIFO	LPT	SPT	SLPT	SSPT	HRRN	WT	HLF	HLFET	CG	DCP	Crit. 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	-



(a) Slowest Frame (b) Delayed Frames (c) Cumulative Slowdown

Figure 6: Average values for all metrics for schedulers using sorted subtasks on different numbers of resources.

If we focus our attention on strategies LPT, WT, CG, and DCP, we can

verify that their improvements over FIFO are all statistically significant (p-values < 0.05), with the exception of the DF metric for LPT (p-value $= 0.73$). WT leads to the same average SF as CG and DCP (p-values $= 0.52$ and 0.23, resp.) while differing in the other metrics. Also, CG and DCP results cannot be differentiated (p-values > 0.05), which contrasts with the results in Scenario I.

Table 3 also shows that the performance gap to the Critical Path is much smaller than before, even though these results use 12 resources only. A better visualization for different numbers of resources can be seen in Fig. 6. The best schedulers here (WT, CG, and DCP) show trends similar to before, as they create a gap between their performance and the baseline that decreases when many resources are available. Yet, in this situation, the absolute differences in values are not strictly decreasing anymore, as FIFO seems to benefit more from the sorted subtasks for small numbers of resources. For instance, comparing Figs. 6b and 4, we can see that changing the micro-scheduler reduces DF on 4 resources by about 5 frames for FIFO (from 138.26 to 132.9) but only 3 for CG (from 131 to 127.9). This effect later disappears when more resources are available. Another difference from Scenario I comes from the fading gap between the best schedulers and the Critical Path. If we consider CG running over 16 resources, the average differences are 0.09, 0.82, and 3.97 for the SF, DF, and CS metrics, respectively. The proximity of these results to the optimal solution highlights 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 60fps can be achieved under the worst load situations. We investigate additional means to improve performance in our final scenario.

5.4 Scenario III - subtask splitting

Given the near-optimal performance of the modified game engine scheduler, the only way to achieve further improvements requires a new optimal. That, in turn, demands changing the task graph. We have identified the two tasks with the longest processing times and changed them to increase their parallelism. For each of their subtasks, we run two subtasks, each with half of the original processing time. This local transformation has no impact on the global task graph nor to the total processing time of the tasks, and it does not affect the majority 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 new performance results are summarized in Table 4. The additional parallelism leads to improvements for all scheduling strategies and metrics. When compared to Scenario I, SF is decreased by about one quarter, DF is reduced by over one half, and CS is reduced by about three quarters. When comparing FIFO's results in Scenarios II and III, these metrics are improved by factors of 1.174, 1.602, and 2.390, resp., which are proportionally larger than the improvements seen from Scenario I to II.

When comparing the algorithms to FIFO, their general behavior remains the same. For example, WT, CG, and DCP show better results than FIFO (p-values < 0.05). WT performed better than DCP (p-values < 0.05), but it performed the same as CG for metrics SF and DF (p-values $= 0.31$ and 0.06, resp.).

The additional parallelism increases the gap between the algorithms and the

Table 4: Average metrics over 12 resources with sorted subtasks and additional parallelism in two tasks. Reductions are calculated in comparison to Table 2.

	FIFO	LPT	SPT	SLPT	SSPT	HRRN	WT	HLF	HLFET	CG	DCP	Crit. 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	28.30	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

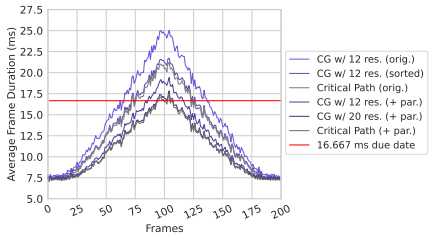


Figure 7: Average duration of each frame for CG and the Critical Path on different numbers of resources and scenarios.

new Critical Path. In general, the best performing algorithms running over 12 resources still show absolute differences of ≈ 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 is not enough to profit from the extra parallelism. Meanwhile, when using 20 resources, these differences are reduced to 0.2, 1.3, and 4.0, respectively. This evolution can be visualized in Fig. 7, which shows the change in average frame duration throughout the simulations for the three scenarios (similarly to Fig. 5). The first gap between CG and the Critical Path is overcome just by sorting subtasks, while the new gap requires using more resources. Overall, we can clearly see that the improvements brought in each scenario makes the game engine more robust to high loads, leading to a better gaming experience.

6 Conclusion and Future Work

In this paper, we have examined the scheduling problem of game engines. Using as a case study a game engine extracted from a modern Ubisoft video game, we have modeled the problem, chosen and adapted scheduling algorithms, and ran an extensive experimental evaluation with an in-house simulator. Compared to the original FIFO scheduler on 12 resources, the use of well-adapted algorithms improved the proposed metrics of Slowest Frame, Delayed Frames, and Cumulative Slowdown up by factors of 1.015, 1.058, and 1.096, resp. The proposed change to the micro-scheduler increased these gains to factors of 1.148, 1.480, and 2.011, with near-optimal results when using more resources. Finally, the additional parallelism in two tasks led to total improvements by factors of 1.354, 2.561, and 5.222.

These results establish the potential contributions that well-adapted scheduling algorithms (local and global, online and offline) and techniques can have on the video game industry. Further research should be dedicated to see how these

results extend to other game engines, video games, and even other interactive simulations. An implementation of the algorithms and techniques in an actual game engine would enable an evaluation of the overhead of run time profiling, online algorithms, and the management of the priority queue. Finally, the effects of hardware heterogeneity (both for uniform and unrelated resources) remains to be studied.

References

[1] Bilge Acun, Akhil Langer, Esteban Meneses, Harshitha Menon, Osman Sarood, Ehsan Totoni, and Laxmikant V Kalé. Power, reliability, and performance: One system to rule them all. *Computer*, 49(10):30–37, 2016.

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

[3] Bret Alfieri. Intel games task scheduler. `https://github.com/GameTechDev/GTS-GamesTaskScheduler`, January 2019. Accessed: 2022-01-04.

[4] Olivier Beaumont, Lionel Eyraud-Dubois, and Yihong Gao. Influence of tasks duration variability on task-based runtime schedulers. *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 16–25, 2019.

[5] Anne Benoit, Louis-Claude Canon, Redouane Elghazi, and Pierre-Cyrille Héam. Update on the Asymptotic Optimality of LPT. In Leonel Sousa, Nuno Roma, and Pedro Tomás, editors, *Euro-Par 2021: Parallel Processing*, pages 55–69, Cham, 2021. Springer International Publishing.

[6] Xiaoqiang Cai, Xianyi Wu, and Xian Zhou. *Optimal stochastic scheduling*, volume 5. Springer, 2014.

[7] Edward G Coffman and Ronald L Graham. Optimal scheduling for two-processor systems. *Acta informatica*, 1(3):200–213, 1972.

[8] Gaoyang Dai, Morteza Mohaqeqi, and Wang Yi. 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)*, pages 119–128, 2021.

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

[10] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[11] Jason Gregory. *Game engine architecture*. Taylor and Francis Ltd., 3 edition, 2018.

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

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

[14] Mordor Intelligence. Global Gaming Market - Growth, Trends, Covid-19 Impact, and Forecasts (2022–2027). `https://www.mordorintelligence.com/industry-reports/global-gaming-market`. Accessed: 2022-01-26.

[15] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC press, 2004.

[16] Flávia Maristela S. Nascimento and George Lima. Effectively scheduling hard and soft real-time tasks on multiprocessors. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 210–222, 2021.

[17] Dan Trietsch, Lilit Mazmanyan, Lilit Gevorgyan, and Kenneth R Baker. Modeling activity times by the parkinson distribution with a lognormal core: Theory and validation. *European Journal of Operational Research*, 216(2):386–396, 2012.

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

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