universität innsbruck

Applying Causal Profiling to Video Games

Moritz Perschke, 11908906
Innsbruck, Januar 2024

Bachelorarbeit

eingereicht an der Universität Innsbruck, Fakultät für Mathematik, Informatik und Physik zur Erlangung des akademischen Grades

Bachelor of Science

Bachelorstudium Informatik
Betreuer:in: Dr. Peter Thoman
Institut für Informatik
Fakultät für Mathematik, Informatik und Physik

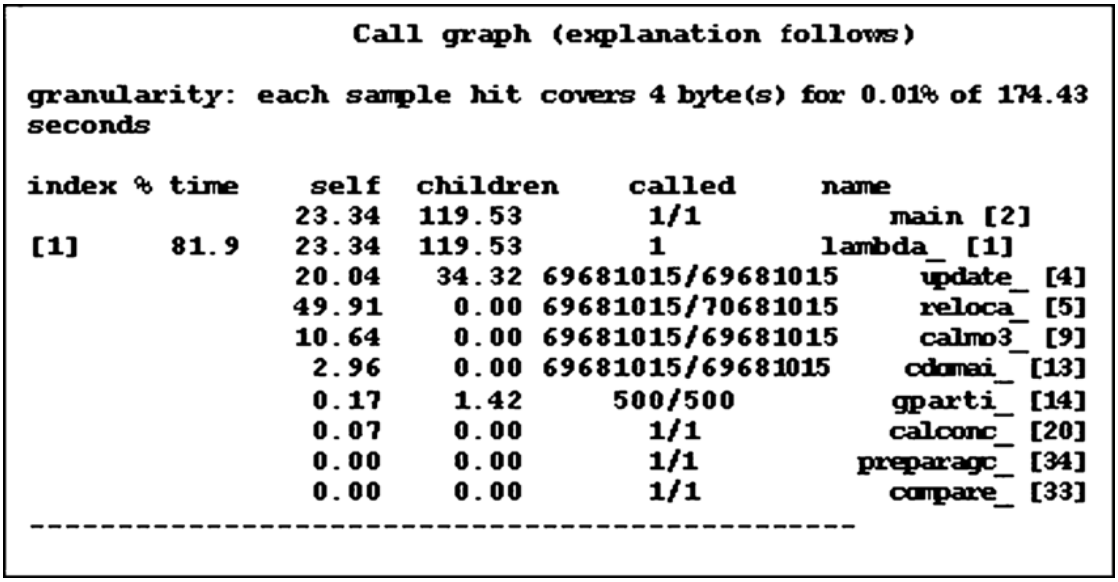
Abstract

Profilers are a useful tool to analyze a program's performance. While there are many different aspects to this, runtime is often regarded as the most important performance metric. Especially in real time applications such as video games, it must be possible to render a new frame many times per second. Profilers such as Superluminal or Optick exist to help a programmer improve the performance of a game. They will usually utilize sampling and trace based profiling, which both measure the execution times of a program's functions.
These measurements are then visualized in a graph to provide a different view of the programs behaviour. In this view, the methods are often ranked by their total time taken over the course of the profiling run. A person tasked with optimization can then use this to reduce the runtime of functions in order to improve overall performance. But optimizing code that takes a long time will not necessarily have the biggest impact on total runtime. Optimizing slow code will often be easier and have a larger impact than optimizing code that is run a lot. Compare implementing a dynamic programming approach in a fibonacci calculation with attempting to reduce time taken by addition.
Causal profiling attempts to find the places in code where optimization will have a large impact by running performance experiments on the target rather than just measuring it. Performance experiments simulate optimizing a method by an arbitrary amount by slowing down every other method.
In this thesis, a new profiler utilizing causal profiling is implemented in order to improve the performance of commercial video games by running performance experiments on the methods provided by the DirectX11 graphics API. The profiler is launched with the game and using the process id injects a Dll which replaces the addresses of the API methods in memory with modified versions. This process is called function hooking. The hook functions then slow down select parts of the DirectX11 API to facilitate virtual speedup.
Five different commercial video games are used to show the strengths and weaknesses of the new profiler. To produce meaningful results, the used game needs to be in a CPU-bound state. If that is the case, the results show that optimizing the functions *DrawIndexed* and *DrawIndexedInstancedIndirect* could theoretically result in an increase in FPS by up to 25% and 15% respectively.
The resulting profile is also compared to a mocked up flat profile in order to show that causal profiling guides optimization effort in a more effective way than traditional methods. This mocked flat profile ranks the methods *Flush* and *CreateTexture2D* as the highest, purely based on their runtime. The causal profile on the other hand does not deem them as relevant, which indicates optimization would have a minimal impact on overall performance.
Necessary further research would be validating the theoretical findings through alterations to the DirectX11 source.

Contents

1 Introduction

Profiling has long been an integral part of software development in order to achieve the highest possible performance. Therefore, a lot of different profilers have been created to achieve the goal of increased performance through different approaches. Profilers like Gprof [11] are used in the pursuit of increasing performance in the sense of faster execution time by measuring every execution of every method and visualizing the result. Valgrind [20] provides tools to find issues with a programs memory performance.
The mentioned profilers, as well as most others, have one common issue when used in an attempt to increase a programs performance. This issue being that all of these profilers only analyze the code as is, without providing any further guidance. A profiler will often only give another view of the codes inner workings without actually showing where optimizing one component will have an impact on the perfomance of the whole program. Or as Curtsinger and Berger put it: "[...] optimizing code that draws a loading animation during file download will not make the program run faster" [7].
To address this issue and create a profiler that will help guide a programmer to where optimization efforts will have the biggest impact, causal profiling was introduced by Curtsinger and Berger in 2015 [7]. Causal profiling uses a series of performance experiments to maximize a given performance metric. This is further explained in section 3.
Frames per second (FPS) is a metric commonly used in video games to characterize performance. This is because a lower time to calculate the color of every pixel in the games window results in more FPS. In other words, the throughput of a real time graphical simulation can be characterized by it's average frame time. To apply causal profiling to video games requires building a new profiler which uses FPS as its default metric for performance.
Modern games that are reasonably complex are usually built from different components and libraries. Most of these components will have an impact on the possible frame rate and are therefore possible targets for optimization. Some games will require a lot of complex logic (e.g. to make decisions for the non-player controlled characters) and can therefore be limited by the calculations done on the CPU. Other games are rather graphically intense due to a high level of detail or a large rendered area (e.g. a game set in a open world with explorable mountains in the distance). In this case the limiting factor will be the GPU. Due to commercial video games being closed source, targeting the implementation of logic would prove difficult.
While implementing the logic themselves, game creators often use an application programming interface (API) to handle the rendering of frames, also called a graphics API. These graphics APIs provide a number of different methods which are often called many times per frame. They are also well documented and publically available. For these reasons the new profiler has been designed to target DirectX11, one of the more commomly used graphics APIs. It will attempt to find potential improvements in frame



```
                    Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.01% of 174.43
seconds

index % time    self  children    called     name
                23.34  119.53       1/1           main [2]
[1]     81.9    23.34  119.53         1         lambda_ [1]
                20.04   34.32 69681015/69681015     update_ [4]
                49.91    0.00 69681015/70681015     reloca_ [5]
                10.64    0.00 69681015/69681015     calmo3_ [9]
                 2.96    0.00 69681015/69681015    cdomai_ [13]
                 0.17    1.42     500/500         gparti_ [14]
                 0.07    0.00       1/1           calconc_ [20]
                 0.00    0.00       1/1          preparagc_ [34]
                 0.00    0.00       1/1           compare_ [33]
-----------------------------------------------
```

Figure 1: Example output generated by gprof [11], taken from [27]

time by running performance experiments on the methods provided by DirectX11. The implementation is described in section 4 while the results are discussed in section 5. The concept of causal profiling is further explained in section 2.

2 Related Work

There exist two main techniques for profiling an application. Trace based profiling uses user-defined marks in the source code to analyze specific parts of the program during execution. These marks are usually inserted before and after a section of code deemed interesting by the programmer and are used to measure time spent in between. This allows profilers like perfetto [10] or dtrace [1] to provide fine-grained measurements on selected targets. For an example of perfetto's output see figure 2
Sampling based profiling interrupts the running process at predefined intervals and inspects the current function call stack. Based on the collected samples the profiler then calculates the amount of time spent in any given function and how many times the function was called. In contrast to the trace based approach, this does not require manual annotation of the source code. But since all sections are treated equally by this technique, short-term spikes (e.g. in runtime) can be missed. Gprof [11] or the linux perf events [13] are examples of sampling based profilers. An example gprof output can be seen in figure 1.

While both of these approaches can generate useful information on their own, most modern profilers like Tracy [30], Optick [28] or Superluminal [26] combine both in a

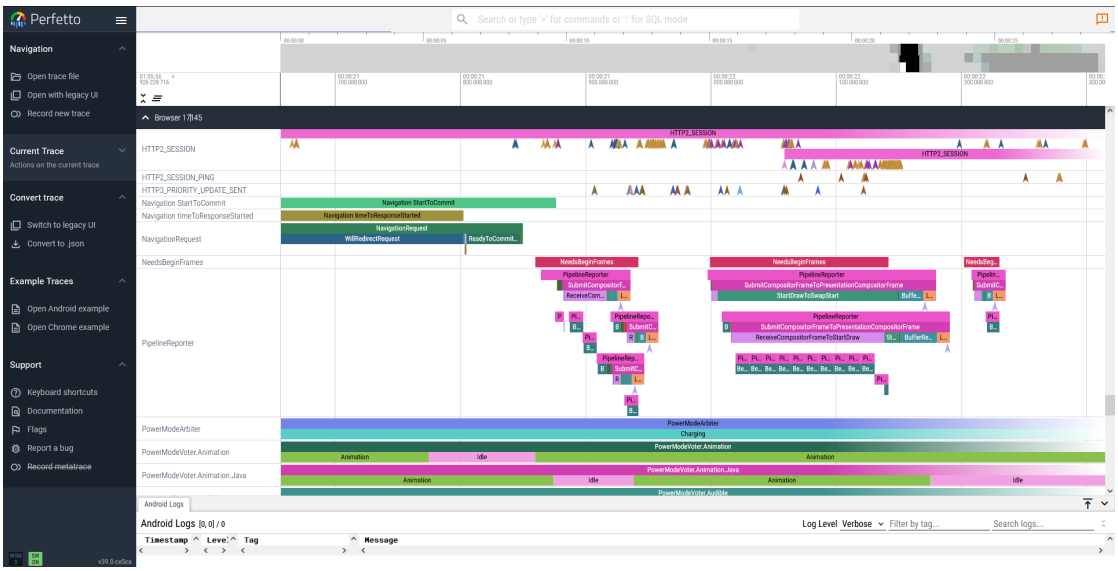


Figure 2: Example output generated by perfetto [10] using the provided chrome example

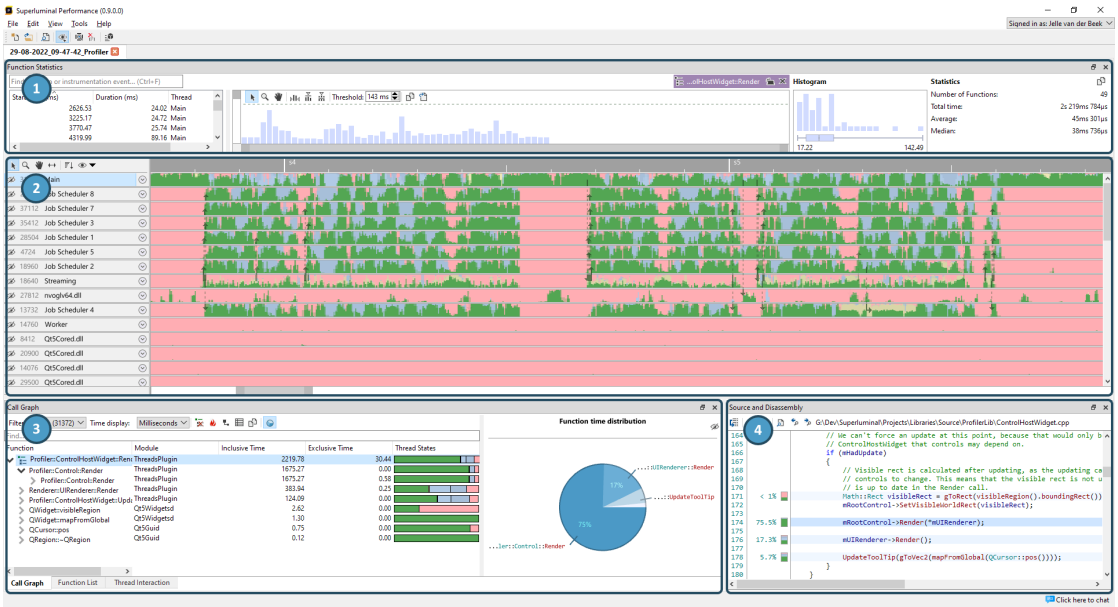


Figure 3: The Superluminal UI as seen in the documentation [29]

hybrid profiling approach. As an example for a hybrid profiler, a screencapture of the Superluminal interface is shown in figure 3. Here it can be seen that in comparison to a profiler using only one of the two techniques, a hybrid profiler provides multiple different views on the programs behaviour. In the example, while others are similar, the information is shown in different panes. The panes visualize:

1. Information about the selected trace, similar to the output of a trace based profiler
2. Information about all concurrent threads
3. A call graph similar to the one generated by a sampling based profiler
4. A view of the corresponding source code

By applying trace and sampling based profiling at the same time, the advantages of both can be leveraged. While often more useful than a profiler utilising only one of the approaches, hybrid profilers still do not guide the optimization effort further than providing a more complete view of the program's behaviour

3 Background: Causal Profiling

3.1 Overview

Causal profiling seeks to mitigate the common problem of misguided optimization efforts by finding code where optimization has an impact instead of finding the longest running lines or functions. It was introduced by Curtsinger and Berger in 2015 [7]. In the same article they also implement it in a linux-based profiler (COZ) and prove its efficacy in different applications. Instead of simply measuring the execution time of methods, COZ runs a series of *performance experiments*.
A performance experiment is defined by a combination of *virtual speedup* and a single target. COZ targets single lines of code. During the execution of a performance experiment, the profiler will sample the instruction pointer of every launched thread to virtually speed up the threads running the targeted line. The concept of virtual speedup was also introduced by Curtsinger and Berger and is further explained in section 3.2. Results are then gathered based on a user defined metric. This metric can be the either the latency or throughput of *progress points* placed by the programmer [7].
Video games provide a clear performance metric through frames per second and frame time while usually relying on the usage of a graphics API such as DirectX, OpenGL or Vulkan to render visuals. To apply causal profiling to video games would mean to find code to optimize in order to increase the framerate. The interface between the GPU and the CPU in the form of DirectX11 has been chosen as a target to find theoretical

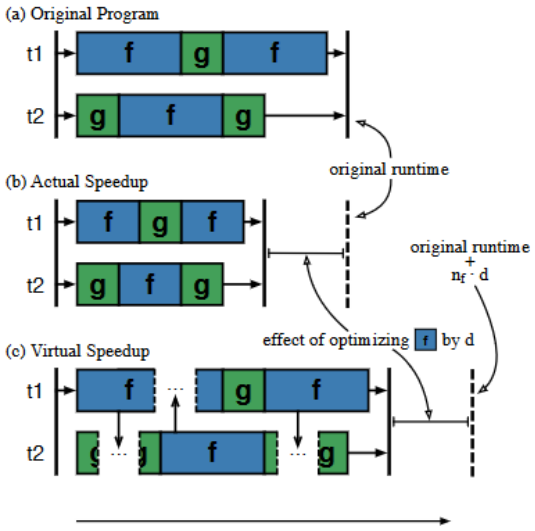


Figure 4: Illustrating the concept of virtual speedup [7]

improvements in framerate by optimizing single functions. The profiler described in section 4 is implemented using the metric of FPS and targets the methods of the DirectX11 API.

3.2 Virtual Speedup

Since actually speeding up a chosen line of code by an arbitrary amount is not actually possible, causal profiling utilizes *virtual* speedup. This idea is best described visually and shown in figure 4. A target (e.g. a line of code or a function) is virtually sped up by inserting pauses in other potential targets while they are not being profiled. COZ [7] implements this through sampling the instruction pointer of any started thread and delaying others based on how often the targeted line is being run. Whenever the targeted line is found in a threads instruction pointer, all other threads are signalled to pause. For example in figure 4 the function f is being analyzed for potential performance increase. When function f is run on one thread, the others execution is paused for a set amount of time based on $d\%$ of f's original runtime. This means that relative to the new overall time taken by the program, function f is faster by $d\%$.

4 Implementation of new profiler

The functionality of the profiler is implemented in two components. A commandline program implementing the logic and calculations (referred to as profiler) and a Dynamic Link Library to manipulate and measure the DirectX11 API methods (referred to as

dll). This dll is injected into a running game and hooks the API's methods (see 4.1). The control flow is implemented as an enum of different statuses set by the profiler in shared memory to define the current action of the dll. Every status is active for 500 in game frames, i.e. 500 calls to the DirectX11 present method [19]

4.1 DLL injection and function hooking

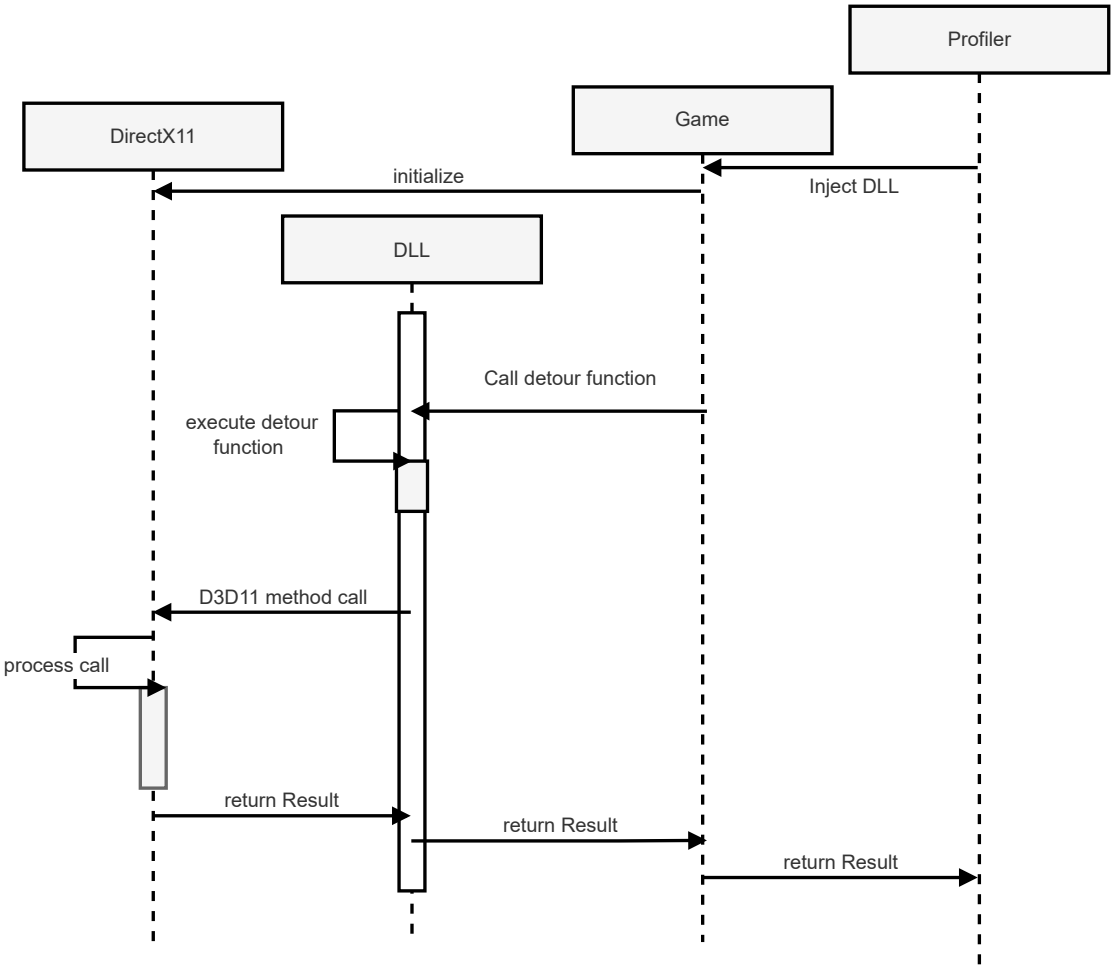


Figure 5: A simplified overview of the interaction between the profiler, the game and DirectX11

As described by Berdajs and Bosnić: "Dll injection is a concept of loading code into the address-space of the target application through a dll, making subsequent interactions with the application's memory and functions easier." [3] In this work, dll injection is done using the *CreateRemoteThread* variant, outlined by Richter et. al. in "Windows

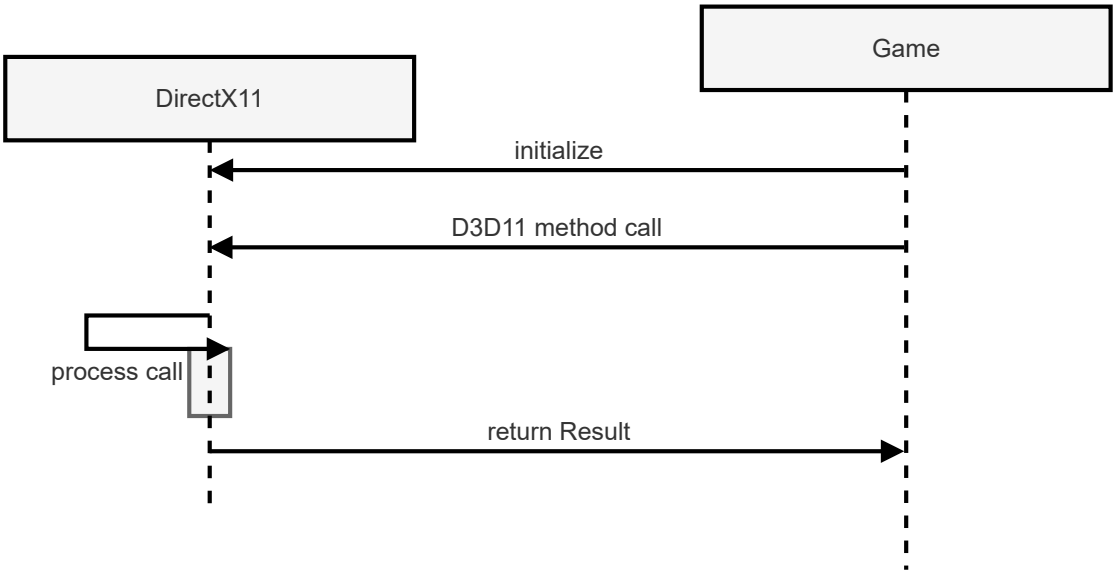


Figure 6: A simplified overview of the interaction of a process with DirectX

via C/C++" [25]. To inject the custom library, first get a handle to the target process using it's process id (PID). After writing the total path of the dll to the process' memory, use *CreateRemoteThread*, passing the PID as parameter, to create a thread in the target process to execute *LoadLibraryW* which loads the custom dll into the remote process. An example for this can be found in the appendix in figure 21.
Since executing code in another processes address space is often done by malware [31] [2], most antivirus software will block and quarantine any process attempting to do so by default [4]. Therefore, an exception in Bitdefenders "Advanced Threat Control" had to be made in order to run the profiler.
Berdajs and Bosnić go on to describe function hooking as follows: "Function hooking, also referred to as code or API hooking, is an approach used to modify an application's behavior by making it use as an arbitrary function instead of the originally intended one." In practice, function hooking replaces the address of a function in a process' memory space with the address of a user defined function to have it call the user defined function instead of the intended one. The profiler uses the open source library Kiero [24] to do so. Kiero is created specifically to apply function hooking to graphics APIs.

4.2 Custom dynamic link library

The dll is intentionally kept as light as possible in order to minimize the introduced overhead . Aside from communication, it implements three different actions for all D3D11 Methods:

- Measuring the duration of execution
- Collecting a unique ID of the calling thread
- Delaying execution by a given number of nanoseconds

As mentioned above, function hooking replaces the address of a function in the memory space of a process with a user defined function. This means that a detour function needs to be defined to replace the original function. These functions are identical for every DirectX11 method.
The detour function takes one of the actions defined above and calls its original. To limit repetition, they are generated at compile time through X-Macro expansion. The only exception to this is the method present, which swaps the front and back buffers to show the new frame as soon as it has finished rendering.
Since the present method is called at the end of rendering a frame, measurement of rendertime and communication is done within the corresponding detour function. The execution time of the present method is not taken into account when measuring frame time. All data collected is written to shared memory for the profiler to process (see 4.3)

4.3 Command line program

The program is given an identifying string (e.g. the name) and the Process Id (PID) of the game to target as command line arguments. The string is used as the name of the directory to write files to. As described in section 4.1, the PID is used for injecting the dll. Two shared memory segments are created using *boost::interprocess* [9] and the Win32 API. The profiler implements two datastructures using the same library, which are able to expand or shrink within shared memory as needed. The implemented datastructures are *Vector<T>* and *Map<K, V>*. From these templates, three different instantiations are defined:

(a) *Map<int, Vector<Nanoseconds>>*

(b) *Map<int, Vector<ThreadIdHash>>*

(c) *Map<int, Map<int, Vector<Nanoseconds>>>*

The map (a) is used to store the execution times of a method, mapping an identifying integer m to a vector of single execution times. (b) is implemented in a similar fashion, mapping m to a vector of Thread Ids. Thread IDs are the *std::thread::id* of a thread, converted to unsigned long long using *std::hash*. Lastly, (c) maps m and a delay x to a vector of Nanoseconds. This map is used to store the resulting frame time when virtually speeding up method m by x percent.

The profiler also creates a *std::deque* internally. This *std::deque* is filled with all combinations of m and x

$$(m, x),\ m \in \{methods | average\_duration > 1000ns\},\ x \in \{0, 0.1, ..., 0.9\}$$

which is then shuffled using a Mersenne Twister (MT) engine [14], to introduce the random element, and saved for later use.
After initialization, the profiler signals the dll to start measuring the execution times of methods. The dll saves the measurements at first locally and only writes them to shared memory after 500 frames, as opening and writing to shared memory during measurement would introduce too much overhead. When the dll signals completion, the profiler reads the values and fills the *std::deque*.
Next the profiler instructs the dll to collect information about the different threads calling a method. The detour function of a method will call *std::this_thread::get_id* and add this id's hash to the map (b).
When all preliminary data is collected, the profiler will start collecting results. First, all methods are delayed by 0% to 90% in 10% increments to establish a baseline. Then the first item of the *std::deque* is removed and the profiler calculates the required delay d_i for every method other than m based on the values e_i in (a).

$$d_i = e_i * x \quad \forall i \in \{methods | i \neq m \wedge e_i > 1000\}$$

This is only done for methods with $e_i > 1000$ because of the $100ns$ accuracy of the underlying clock. Due to this accuracy, methods where the shortest delay would be less than $100ns$ are not delayed at all (see section 4.4). These delays are inserted into an array and sent to the dll, which applies them to the corresponding method based on the array index to virtually speed up m. The resulting frame time is measured for 500 frames and written to the map in shared memory (c). A visualization of this sequence can be found in figure 7.
When no more combinations remain in the *std::deque*, the profiling run is complete and the profiler sets all delays to zero. All results are then read from shared memory and written to Json objects using the *nlohmann/Json* library [21]. Files are created for

- Frame rates (*Framerates.json*)
- Frame times (*Frametimes.json*)
- Durations of methods (*MethodDurations.json*)
- Thread IDs (*ThreadIDs.json*)

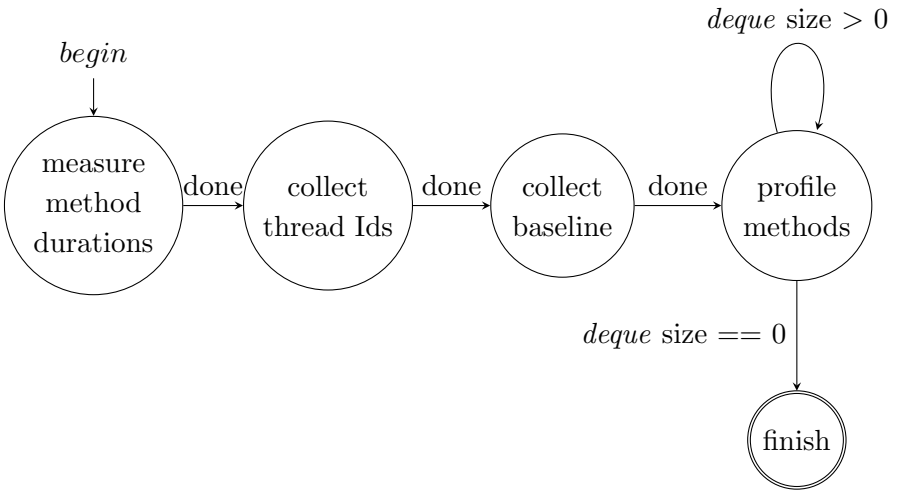


Figure 7: Visualization of the different states of the profiler

Frame times represent the duration of rendering a frame while frame rates represent the duration of the present method in addtion to rendering. The *MethodDurations.json* file contains all measurements of time taken by the DirectX11 methods while the *ThreadIDs.json* lists the hashed thread id of a calling thread for every method. After writing these files the program exits.

4.4 Time measurement

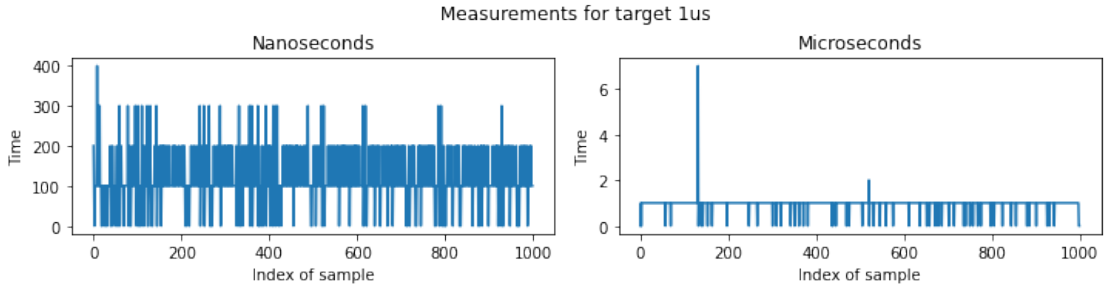


Figure 8: Accuracy comparison of Nanoseconds(target 100) and Microseconds(target 1) using *std::chrono::high_resolution_clock*

On a 144hz refresh rate monitor, framtimes are regularly < 6.9 miliseconds. To measure all methods called within this time, the timer with the highest resolution possible has to be used. The C++ standard library provides a *std::chrono::high_resolution_clock* struct in the *<chrono>* header [16].
On Windows this method calls the Win32 API function *QueryPerformanceCounter*

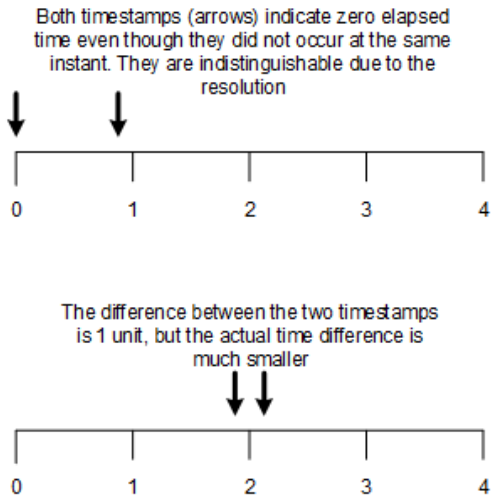
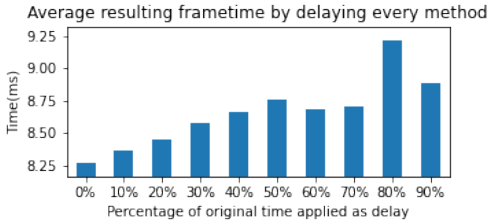
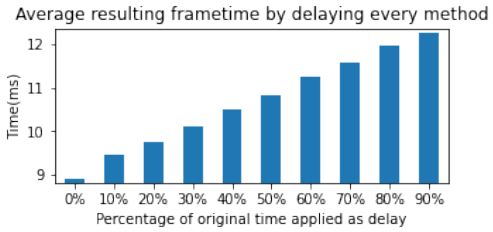


Figure 9: Visual respresentation of tick uncertainty, taken from [15]



(a) Results after taking all measurements



(b) Results after discarding 100ns measurements

Figure 10: Comparison of discarding/-keeping 100ns measurements

(QPC) [17], which according to Microsoft is "typically the best method to use to timestamp events and measure small time intervals[...] [15]. Evaluating QPC on the machine described in section 5.1, the underlying hardware counter appears to have a tickrate of 100ns. When choosing nanosecond precision for the profiler, around 25% of measurements for a given method result in a measured time of 100ns. As can be seen in figure 9 any actual time t taken where $0ns < t < 100ns$ will be counted as a single tick. Therefore, any measurement of 100ns has a possible error anywhere between 1% and 99% and is discarded. For a measurement of 200ns, this error shrinks to 0.5% and 50%, which is not large enough to negatively impact the results of the profiler. The impact of this error can be seen in figure 10. In these graphs, the average frame time when delaying all methods is shown on the Y-Axis. The X-Axis represents the percentage of a methods original duration applied as delay. When all delays are increased by the same amount, one would expect the resulting frame time to increase in a close to linear fashion. In figure 10a), this is not the case. Instead, the averages don't steadily increase but increase and decrease randomly. Whereas 10b) shows how the frame rates increase as expected. In figure 10b, as in the results shown in section 5.2, measuremens $< 100ns$ are discarded.

4.5 Sleep method

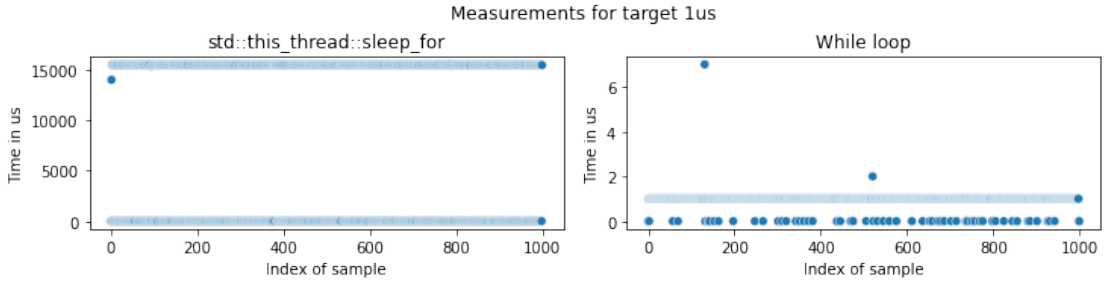


Figure 11: Comparison of sleeping using *std::this_thread::sleep_for* and a while loop. Shown are the measurements of calling *std::this_thread::sleep_for* and the implemented sleep function with the goal of 1 microsecond 1000 times.

The C++ standard library provides programmers an easy way to insert a pause into their code. *std::this_thread::sleep_for* and *std::chrono::microseconds* guarantee a delay of the specified amount of time *at minimum* [6]. These methods are defined in the <*thread*> and <*chrono*> headers respectively. It is evident from figure 11 that *std::this_thread::sleep_for* often takes $1.5 * 10^3$ longer than requested in our case. Since sleeping for longer than specified is not viable in this context, a naive sleep function is implemented (see figure 20 in the appendix). This new sleep function utilizes a *while* loop. The condition of this *while* loop is a comparison of the current timestamp aquired by the *std::chrono::steady_clock* and the given delay added to a timestamp aquired at the beginning of the loop. As long as the current timestamp is less than the sum, the loop condition will evaluate to true. If the remaining time is more than one milisecond, the function calls *std::this_thread::yield* to save system resources. If the current time is more than or equal to the sum, the function returns and normal execution resumes.

4.6 Analysis of data

As mentioned in section 4.3, measurements are written to a json file in the projects *data/* directory. This data is then imported into a Jupyter notebook for further analysis.
Accurate time measurement in the nanosecond range is difficult and inaccurate (see 4.5), causing the variance between single measurements to be quite high. Scheduling conflicts will cause the frame time to spike at seemingly random intervals due the game waiting on system resources used by other processes, causing outliers. Outliers are removed using the Interquartile range method. This method calculates the 25th and 75th percentile, $Q1$ and $Q3$ respectively, and removes data points that do not fit within a a range calculated from these values.

$$IQR = 1.3 * (Q3 - Q1)$$

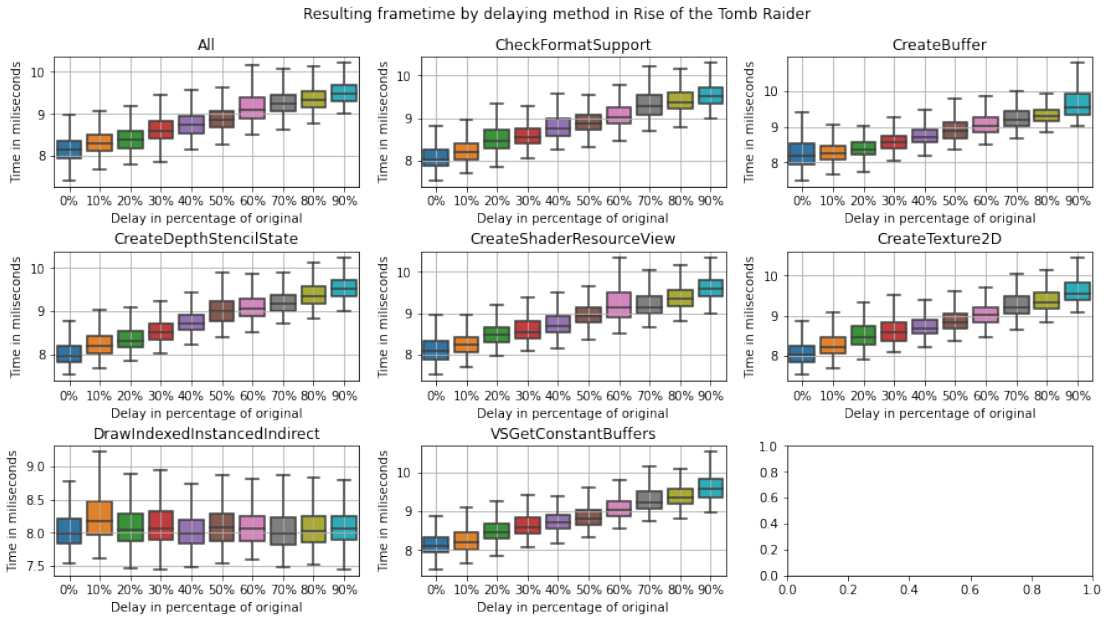


Figure 12: An example of plotting all measurements in a boxplot. Data is collected using Rise of the Tomb Raider

$$Q1 - 1.5 * IQR < x < Q3 + 1.5 * iqr$$

This is a common statistical tool to remove outliers and applicable in this case as there are enough measurements to warrant the use of this method. The average frame time of every combination between method m and virtual speedup x (R_{mx}) is then calculated using numpys *mean* method and collected in a Pandas DataFrame [23].
Multiple plots are created for evaluation. One line- and one boxplot of every collected frame time, grouped by method and delay (see figure 12 for an example). This plot is useful to ensure that the required steps were taken properly and the produced data is useful. The different lines or boxes in these graphs should have a clear separation between the different applied virtual speedups, especially when the delays are applied to all methods. As it is the best indicator for a correctly executed profiling run, the average frame time after delaying all methods (A_x) is plotted again in a seperate histogram, similar to figure 10a or 10b.
Results are calculated using A_x as baseline. The difference y between the baseline and the average R_{mx} for a run is calculated and plotted absolute and relative to its baseline.

$$y = abs(A_x - R_{mx})$$

These y values relative to A_0 represent the potential percentage increase in FPS while the absolute represents the potential reduction in frame time.

5 Evaluation

5.1 Overview

CPU	AMD Ryzen 7 5800X 8-Core 4.2 GHz
GPU	NVIDIA GeForce RTX 3080
RAM	32 GB
NVIDIA Driver	537.13
OS	Windows 10 Home Build 19045.3693
Monitor	ASUS VG27AQ1A

Table 1: Hardware used for testing

The newly implemented profiler is used on relatively modern hardware (see table 1). To ensure the profiler measures the performance of the API, the game cannot be limited by the GPU's ability to render frames. Therefore, a CPU-bound state is required. The solution to this is achieved with multiple steps.
By using Windows' *Advanced Power Management*, the maximum processor state can be limited. For the following results the maximum state of the processor has been set to 10%. Through this, we attempt to increase the time needed to process calls to the graphics API. To decrease the load on the GPU, the game is run in windowed mode with the lowest possible resolution.
The used monitor provides Vertical Sync (VSync) capabilities, which has to be turned off. VSync is a technology that locks the frame rate when exceeding the refresh rate of the monitor to combat screen tearing [22]. This renders any data collected unusable, as not the actual frame time is measured but the frame time allowed by VSync on the monitor. The same applies for limiting the possible FPS in game, just that in this case the game does not allow the highest possible frame rate. This setting has to be disabled too.
To find the DirectX11 methods where optimization has the highest impact, commercial video games utilising it for rendering are used. The used games are listed below and are aquired through the steam platform [5]. Games were chosen based on graphical complexity and availability. Furthermore, other games that would fit the criteria often employ anticheat measurements like BattlEye [12] or easy Anti-Cheat [8], which prevents the profiler from injecting the dll.

- Borderlands 3
- Shadow of the Tomb Raider
- Dirt Rally 2.0

Furthermore, the listed games are CPU intensive on their own without the additional steps described above. For comparison, less graphically complex and CPU intensive games are shown.

- Disco Elysium
- Outer Wilds

To verify all of the mentioned steps were taken correctly, the notebook described in 4.6 plots all measurements for R_{mx}. An example graph showing all measurements can be seen in figure 12.
With the game in a CPU-bound state, the profiling results will show a clear separation between the different speedups (boxes in the example figure 12) for some methods. In case the separation is not clear in any of the plots, especially the one labeled *All*, the produced results are not going to be meaningful.

5.2 Results

To collect the following results, the game was brought into a steady state, quit and reopened in the steady state. During reopening the game was injected with the dll for profiling. A state thought to be steady is one where no new assets need to be loaded and the game does not require any input. Enemies attacking the player character (PC) for example would require input to prevent a game over screen. A game over screen is usually very different from the previous state, requiring assets to be loaded or unloaded. Changing the state of the game during a profiling run would make all previously collected data incomparable as it would require rendering different assets. The specific state of each game is described in it's corresponding section. In the chosen state the mouse controlling the PC's perspective was not moved and no keyboard inputs were given during the entire run.

5.2.1 Borderlands 3

To profile DirectX11 using Borderlands 3, an existing save file was loaded and the PC was placed in the games' hub. A screencapture of this hub can be found in the appendix in figure 22. A CPU-bound state was achieved by changing some settings from their default values in addition to the steps described above:

- *Graphics API*: DirectX11
- *Display Mode*: Windowed

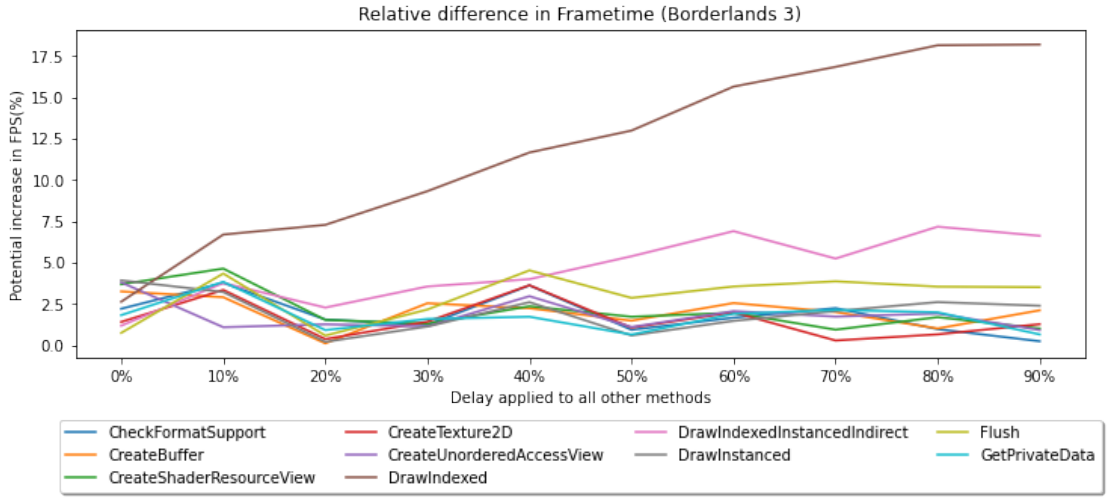


Figure 13: Results for Borderlands 3

- *Resolution*: 1290x720
- *Limit Frame Rate*: Unlimited
- *Graphics Quality*: Very Low

The hub location was chosen as there is some NPC activity and because it is a safe zone for the player. A safe zone guarantees that no enemies will try to attack. NPC activity will increase the load on the CPU, moving the game further into a CPU-bound state. Graph 13 shows that FPS could theoretically be improved up to $\sim$ 18% when optimizing *DrawIndexed* by 90%. *DrawIndexedInstancedIndirect* could also improve FPS by $\sim$ 5% when optimized more than 50%. No other method seems to have a relevant impact.

5.2.2 Rise of the Tomb Raider

The hub in Borderlands 3 is a confined space within a ship where any asset beyond the windows is part of a skybox. To profile DirectX11 with a larger rendered space, an existing save file in Rise of the Tomb Raider was used. The steady state this save file renders a large valley containing many buildings the player can enter. The exact state can be seen in the appendix in figure 23 and is, according to the main menu, "Soviet Installation, 12% complete".
Some of the relevant settings were changed from the default:

- *Fullscreen*: Off
- *Resolution*: 800x600

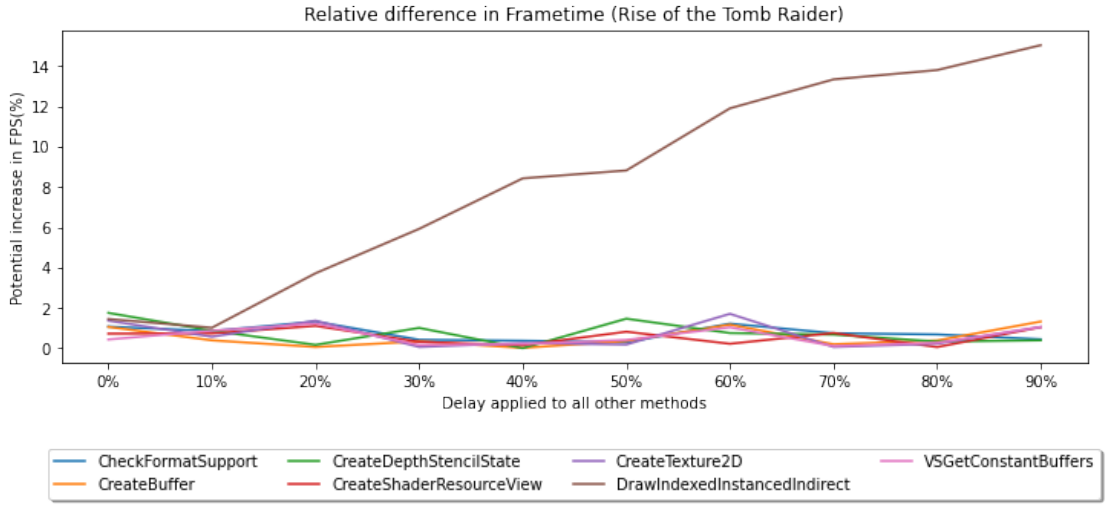


Figure 14: Results for Rise of the Tomb Raider

- *VSync*: Off
- *Anti-Aliasing*: Off
- *Graphics Preset*: Lowest

As can be seen in graph 14, optimizing *DrawIndexedInstancedIndirect* in Rise of the Tomb Raider shows a smaller potential improvement than *DrawIndexed* did in Borderlands 3. Optimizing this method by 90% would improve FPS by only $\sim$ 14%. Similar to Borderlands, no other method shows a relevant improvement.

5.2.3 Dirt Rally 2.0

The steady state chosen for Dirt Rally 2.0 was in a *Free Roam* freeplay with the surface type *Mixed Dirt (DirtFish)* and the selected car was the *BMW E30 M3 EVO Rally*. The camera perspective was chosen to be a following birds-eye view of the car. Relevant setting were changed again:

- *Resolution*: 1024x76
- *Display mode*: Windowed
- *Vsync*: Off
- *Mulitsampling*: Off
- *Anisotropic Filtering*: Off

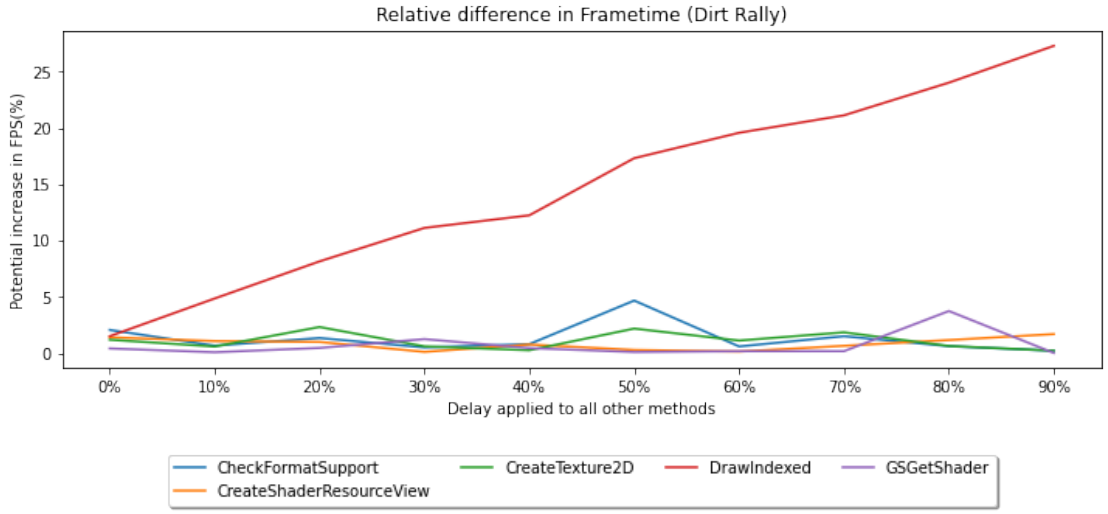


Figure 15: Results for Dirt Rally

- *TAA*: Off
- *Quality Preset*: Ultra Low

Similar to Borderlands 3, Dirt Rally 2.0 shows through experimentation that the highest potential FPS improvement would be optimizing *DrawIndexed* (see figure 15).. Optimized by 90% this method yields an improvement of $\sim 25\%$.

5.2.4 Outer Wilds

As mentioned in section 5.1, Outer Wilds was chosen to represent a game not graphically or CPU intensive enough to produce meaningful results. The settings changed to collect the results shown in figure 16 are as follows:

- *Fullscreen mode*: Disabled
- *Resolution*: 1024x600
- *VSync*: Disabled
- *Anti Aliasing*: None
- *Texture Quality*: Eighth
- *Shadow Quality*: Low
- *SSAO Quality*: Off

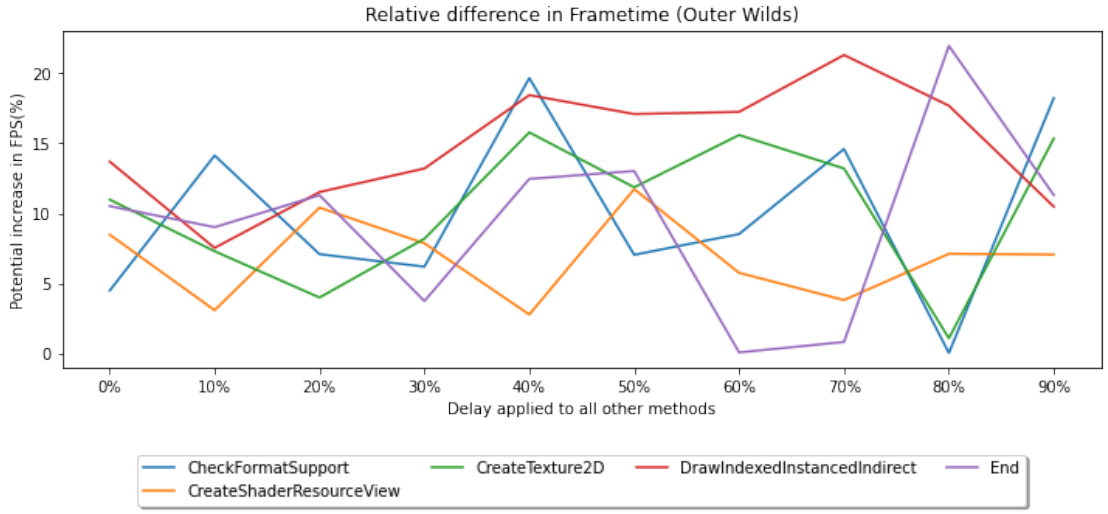


Figure 16: Results for Outer Wilds

- *Water and Lighting Effects Quality*: Low

The profiled methods appear to produce improved frame times at certain optimizations, however one would expect these improvements to follow some sort of linearity. In figure 16 the methods *CreateTexture2D* and *CheckFormatSupport* show an FPS improvement of $\sim 15\%$ at 40% improvement, but drop down to 0% at 80% improvement. The steady state used to profile Outer Wilds can be seen in the appendix in figure 25.

5.2.5 Disco Elysium

To ensure that Outer Wilds is not an exception, Disco Elysium was used for profiling the API as well. To get as close as possible to a CPU-bound state, settings were changed:

- *Display Mode*: Windowed
- *Resolution*: 1280x720
- *Environment FX*: Low
- *Anti-Aliasing*: Off
- *Dynamic Shadows*: Off
- *Shader Quality*: Modest

Once again, the results shown in figure 17 suggest possible speedup between $\sim 1\%$ and $\sim 3\%$ in a nonsensical fashion. The calculated speedup increases and drops at seemingly random points. A screenshot of the chosen steady state is included in the appendix in figure 26.

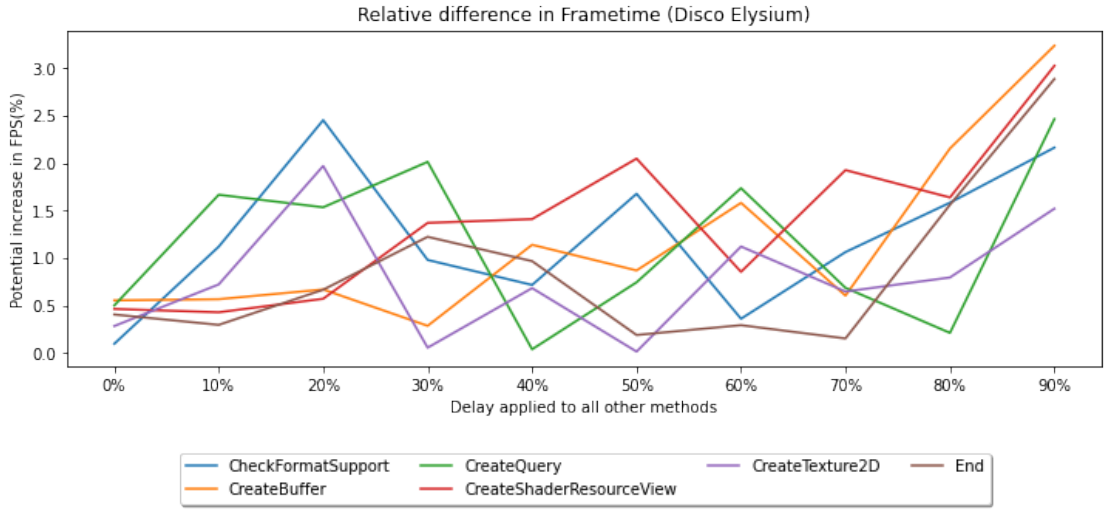


Figure 17: Results for Disco Elysium

6 Conclusion

In this thesis a new profiler using causal profiling to find potentially impactful optimizations in the DirectX11 API was implemented. This profiler leverages dll-injection and function hooking techniques [3] in order to manipulate commercial video games to run performance experiments. Performance experiments simulate the optimization of targets and show their impact on a predefined metric. In the case of the new profiler, frames per second (FPS) was chosen as the metric.

Profiling DirectX11 on three different commercial video games shows that optimizing the methods *DrawIndexed* and *DrawIndexedInstancedIndirect* could yield an increase in FPS of up to 17%. The main goal of causal profiling is to not only provide a different view of the programs behaviour but also to guide the optimizaion effort further than a flat profile generated by traditional profilers. Building a comparable flat profile from the data collected by the new profiler allows for a comparison of the two approaches.

Figure 18 shows such a flat profile built from the data collected over 500 frames. The shown values are the total collected values divided by 500 in order to represent the methods called when rendering a single frame. Using this profile in a naive manner would lead a programmer to only consider the relevant methods after attempting to optimize six others.

Ranked highest in the flat profile, the method *Flush* is a good example of why the causal profile marks different methods as important. The methods documentation states that "...Flush operates asynchronously..." [18]. Attempting to optimize an asynchronous function is not going to result in a large improvement on rendering performance, as it does not slow down the working thread. The documentation also states: "If an application

	calls	avg	min	max
Flush	3.006	259.80	65.8	975.2
CreateTexture2D	0.010	173.76	153.2	198.6
CreateBuffer	7.060	31.68	9.4	585.8
CreateUnorderedAccessView	1.002	16.96	12.0	182.2
CreateShaderResourceView	7.024	11.24	2.8	579.2
CheckFormatSupport	0.010	6.40	5.8	7.2
DrawIndexedInstancedIndirect	0.062	5.52	0.4	43.2
DrawInstanced	0.374	2.31	0.4	55.8
DrawIndexed	3.650	2.14	0.4	135.0
GetPrivateData	1.182	2.03	0.4	6.4
VSGetSamplers	1.002	1.17	0.6	18.8
PSGetSamplers	1.002	1.05	0.6	3.2
Release	2.062	1.02	0.4	64.0

.
.
.

Figure 18: A mocked flat profile based on the data collected while profiling Borderlands 3 (truncated)

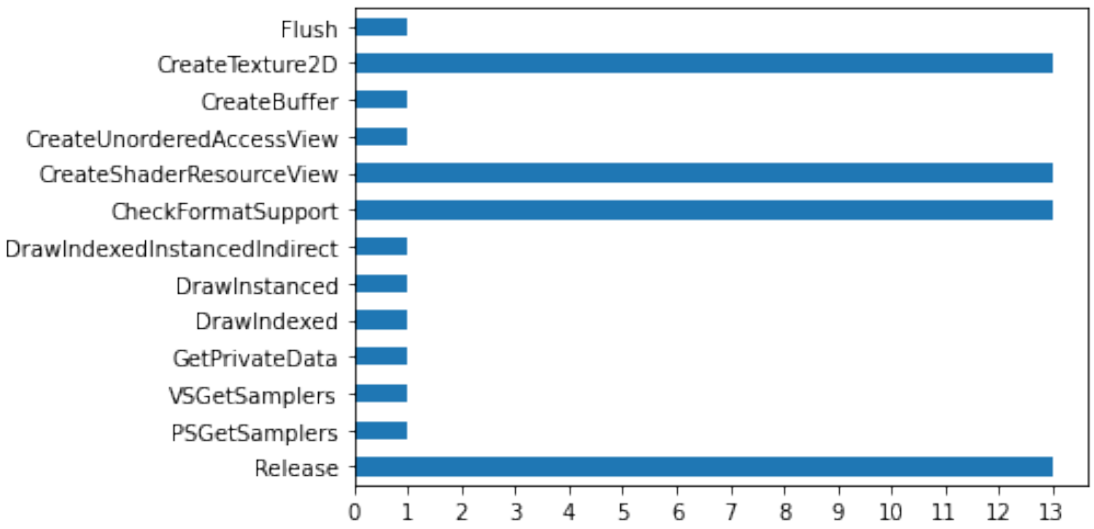


Figure 19: A graphical representation of the amount of Threads calling a method based on the data collected while profiling Borderlands 3 (truncated)

calls this method when not necessary, it incurs a performance penalty." [18] A better approach to reducing the portion of time taken by this method would therefore be to reduce the number of calls instead of the method itself.
The presumption of a single render thread explains why other methods, namely

- *CreateTexture2D*
- *CreateShaderResourceView*
- *CheckFormatSupport*
- *Release*

do not show a large impact in any of the games mentioned above. Figure 19 shows these methods being called by thirteen different threads. The performance impact on FPS of methods called by threads that do not render the next frame is not going to correspond directly to the time they take during rendering, as they do not slow down the working thread.
Due to constraints on time and scope, the theoretical optimizations to the DirectX11 API were not attempted in practice. To further attempt to apply causal profiling could mean not only targeting the interface between CPU and GPU but the source code of a game. The results of this thesis would also need to be validated by optimizations to the *DrawIndexed* and *DrawIndexedInstancedIndirect*.

References

[1] Mike Shapiro Adam Leventhal. Dtrace. Website `https://dtrace.org/`. [Online; Accessed 06.01.2024].

[2] Ahmed Alasiri, Muteb Al Zaidi, Dale Lindskog, Pavol Zavarsky, Ron Ruhl, and Shafi Alassmi. Comparative analysis of operational malware dynamic link library (dll) injection live response vs. memory image. 07 2012.

[3] J. Berdajs and Z. Bosnić. Extending applications using an advanced approach to dll injection and api hooking. *Softw. Pract. Exper.*, 40(7):567–584, jun 2010.

[4] Bitdefender. Advanced threat control. Support Center `https://www.bitdefender.com/business/support/en/77211-376309-advanced-threat-control.html`. [Online; Accessed 02.09.2023].

[5] Valve Corporation. Steam. Website `https://store.steampowered.com`, 2015. [Online; Accessed 06.12.2023].

[6] cppreference.com. std::this_thread::sleep_for. C++ Standard Library `https://en.cppreference.com/w/cpp/thread/sleep_for`. [Online; Accessed 02.09.2023].

[7] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. *CoRR*, abs/1608.03676, 2016.

[8] Epic Games. easy anti-cheat. Website `https://www.easy.ac/en-us/`. [Online; Accessed 09.01.2024].

[9] Ion Gaztañaga. Boost.interprocess. Boost website `https://www.boost.org/doc/libs/1_83_0/doc/html/interprocess.html`. [Online; Accessed 05.12.2023].

[10] Google. System profiling, app tracing and trace analysis. Website `https://perfetto.dev/`. [Online; Accessed 06.01.2024].

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, jun 1982.

[12] BattlEye Innovations. Battleye - the anti-cheat gold standard. Website `https://www.battleye.com/`. [Online; Accessed 09.01.2024].

[13] Linux Kernel. perf: Linux profiling with performance counters. Website `https://perf.wiki.kernel.org/index.php/Main_Page`. [Online; Accessed 09.12.2023].

[14] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, jan 1998.

[15] Microsoft. Acquiring high-resolution time stamps. Windows System Information https://learn.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps#resolution-precision-accuracy-and-stability. [Online; Accessed 03.09.2023].

[16] Microsoft. high_resolution_clock struct. Win32 API Documentation https://learn.microsoft.com/en-us/cpp/standard-library/high-resolution-clock-struct?view=msvc-170. [Online; Accessed 03.09.2023].

[17] Microsoft. Queryperformancecounter function (profileapi.h). Win32 API Documentation https://learn.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter. [Online; Accessed 03.09.2023].

[18] Microsoft. ID3D11DeviceContext::Flush method (d3d11.h). Website https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-flush, 2021. [Online: Accessed 17.01.2024].

[19] Microsoft. IDXGISwapChain::Present method (dxgi.h). Website https://learn.microsoft.com/en-us/windows/win32/api/dxgi/nf-dxgi-idxgiswapchain-present, 2022. [Online; Accessed 05.12.2023].

[20] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[21] nlohmann. Json for modern c++. Github, https://github.com/nlohmann/json. [Online; Accessed 02.09.2023].

[22] NVIDIA. Adaptive vsync. NVIDIA website https://www.nvidia.com/en-gb/geforce/technologies/adaptive-vsync/technology. [Online; Accessed 03.09.2023].

[23] pandas. Python data analysis library. https://pandas.pydata.org/. [Online; Accessed 03.09.2023].

[24] Rebzzel. Universal graphical hook for a d3d9-d3d12, opengl and vulkan based games. Github, https://github.com/Rebzzel/kiero. [Online; Accessed 02.09.2023].

[25] Jeffrey M. Richter and Christophe Nasarre. *Windows via C/C++*. Microsoft Press, USA, 5th edition, 2007.

[26] Jelle van der Beek Ritesh Oedayrajsingh Varma. Profiling, re-invented. Website https://superluminal.eu/, 2017. [Online; Accessed 07.01.2024].

[27] Débora Roberti, Roberto Souto, Haroldo Campos Velho, Gervásio Degrazia, and D. Anfossi. Parallel implementation of a lagrangian stochastic model for pollutant dispersion. *International Journal of Parallel Programming*, 33:485–498, 10 2005.

[28] Vadim Slyusarev. bomboby/optick. Github, 2015. `https://github.com/bombomby/optick` (Accessed: 07.01.2024).

[29] Superluminal. Superluminal documentation. Website `https://www.superluminal.eu/docs/documentation.html#quick_ui_overview`. [Online; Accessed 07.01.2024].

[30] Bartosz Taudul. wolfpld/tracy. Github, 2015. `https://github.com/wolfpld/tracy` (Accessed: 07.05.23, 19:50).

[31] Craig Wright. Taking control, functions to dll injection. *SSRN Electronic Journal*, 01 2007.

Appendix

Code Examples

```
void  little_sleep (DWORD delay) {
auto start = MethodDurations::now();
auto end = start + static_cast<Microseconds>(delay);
while (MethodDurations::now() < end) {
    if ((MethodDurations::now() - end) > std::chrono::milliseconds(1)){
        std::this_thread::yield();
    }
}
}
```

Figure 20: Sleep function implemented for the profiler DLL

```
hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);

rBuffer = VirtualAllocEx(
        hProcess,
        rBuffer,
        sizeof(dllPath),
        (MEM_RESERVE | MEM_COMMIT),
        PAGE_READWRITE
    );
WriteProcessMemory(
        hProcess,
        rBuffer,
        (LPVOID)dllPath,
        sizeof(dllPath),
        NULL
    );
hKernel32 = GetModuleHandle(L"Kernel32");

LPTHREAD_START_ROUTINE loadlib =
    (LPTHREAD_START_ROUTINE) GetProcAddress(
                                hKernel32, "LoadLibraryW"
                            );
hThread = CreateRemoteThread(
        hProcess,
        NULL,
        0,
        loadlib,
        rBuffer,
        0,
        &TID
    );
```

Figure 21: An example how how DLL injection is done

Game Screenshots

Figure 22: Screenshot of the steady state in which Borderlands 3 was used for profiling

Figure 23: Screenshot of the steady state in which Rise of the Tomb Raider was used for profiling

Figure 24: Screenshot of the steady state in which Dirt Rally 2.0 was used for profiling

Figure 25: Screenshot of the steady state in which Outer Wilds was used for profiling

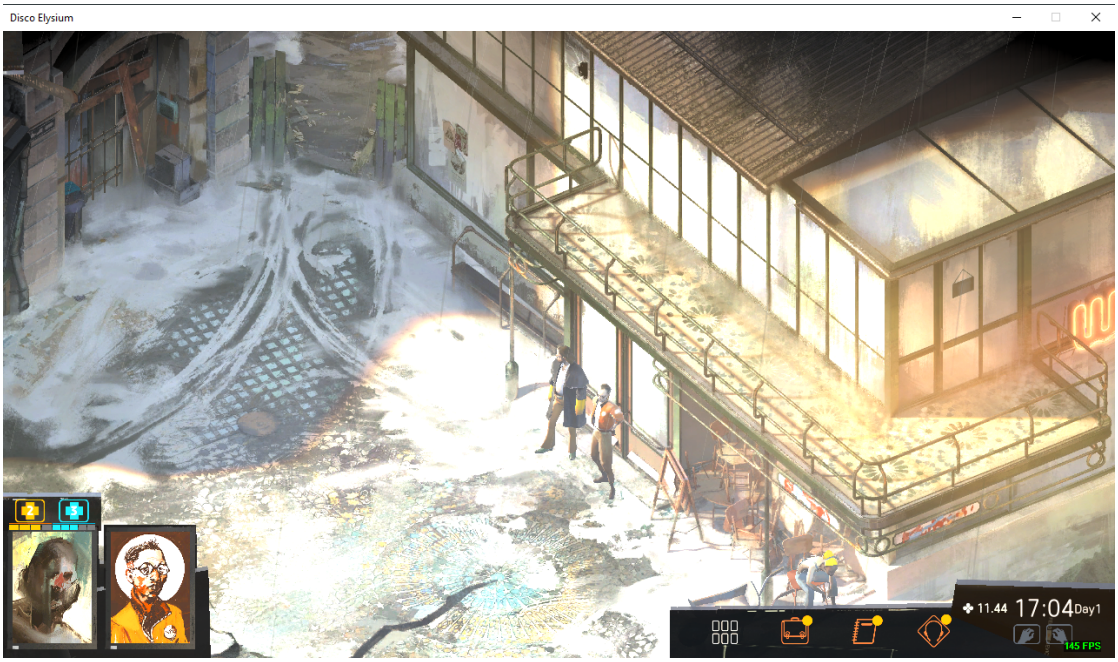


Figure 26: Screenshot of the steady state in which Disco Elysium was used for profiling