Tracy Profiler

The user manual

Bartosz Taudul <wolf.pld@gmail.com>

June 11, 2020

https://github.com/wolfpld/tracy

Quick overview

Hello and welcome to the Tracy Profiler user manual! Here you will find all the information you need to start using the profiler. This manual has the following layout:

- Chapter 1, *A quick look at Tracy Profiler*, gives a short description of what Tracy is and how it works.
- Chapter 2, *First steps*, shows how the profiler can be integrated into your application, and how to build the graphical user interface (section 2.3). At this point you will be able to establish a connection from the profiler to your application.
- Chapter 3, *Client markup*, provides information on how to instrument your application, in order to retrieve useful profiling data.
- Chapter 4, *Capturing the data*, goes into more detail on how the profiling information can be captured and stored on disk.
- Chapter 5, *Analyzing captured data*, guides you through the graphical user interface of the profiler.
- Chapter 6, *Importing external profiling data*, documents how to import data from other profilers.
- Chapter 7, *Configuration files*, gives information on the profiler settings.

Contents

1 A quick look at Tracy Profiler

Tracy is a real-time, nanosecond resolution *hybrid frame and sampling profiler* that can be used for remote or embedded telemetry of games and other applications. It can profile CPU (C, C++11, Lua), GPU (OpenGL, Vulkan, Direct3D 12, OpenCL) and memory. It also can monitor locks held by threads and show where contention does happen.

While Tracy can perform statistical analysis of sampled call stack data, just like other *statistical profilers* (such as VTune, perf or Very Sleepy), it mainly focuses on manual markup of the source code, which allows frame-by-frame inspection of the program execution. You will be able to see exactly which functions are called, how much time is spent in them, and how do they interact with each other in a multi-threaded environment. In contrast, the statistical analysis may show you the hot spots in your code, but it is unable to accurately pinpoint the underlying cause for semi-random frame stutter that may occur every couple of seconds.

Even though Tracy targets *frame* profiling, with the emphasis on analysis of *frame time* in real-time applications (i.e. games), it does work with utilities that do not employ the concept of a frame. There's nothing that would prohibit profiling of, for example, a compression tool, or an event-driven UI application.

You may think of Tracy as the RAD Telemetry plus Intel VTune, on overdrive.

1.1 Real-time

The concept of Tracy being a real-time profiler may be explained in a couple of different ways:

1. The profiled application is not slowed down by profiling[1]. The act of recording a profiling event has virtually zero cost – it only takes a few nanoseconds. Even on low-power mobile devices there's no perceptible impact on execution speed.

2. The profiler itself works in real-time, without the need to process collected data in a complex way. Actually, it is quite inefficient in the way it works, as the data it presents is calculated anew each frame. And yet it can run at 60 frames per second.

3. The profiler has full functionality when the profiled application is running and the data is still being collected. You may interact with your application and then immediately switch to the profiler, when a performance drop occurs.

1.2 Nanosecond resolution

It is hard to imagine how long a nanosecond is. One good analogy is to compare it with a measure of length. Let's say that one second is one meter (the average doorknob is on the height of one meter).

One millisecond ($\frac{1}{1000}$ of a second) would be then the length of a millimeter. The average size of a red ant or the width of a pencil is 5 or 6 mm. A modern game running at 60 frames per second has only 16 ms to update the game world and render the entire scene.

One microsecond ($\frac{1}{1000}$ of a millisecond) in our comparison equals to one micron. The diameter of a typical bacterium ranges from 1 to 10 microns. The diameter of a red blood cell, or width of strand of spider web silk is about 7 µm.

And finally, one nanosecond ($\frac{1}{1000}$ of a microsecond) would be one nanometer. The modern microprocessor transistor gate, the width of DNA helix, or the thickness of a cell membrane are in the range of 5 nm. In one ns the light can travel only 30 cm.

Tracy can achieve single-digit nanosecond measurement resolution, due to usage of hardware timing mechanisms on the x86 and ARM architectures[2]. Other profilers may rely on the timers provided by operating

[1] See section 1.7 for a benchmark.

[2] In both 32 and 64 bit variants. On x86 Tracy requires a modern version of the `rdtsc` instruction (Sandy Bridge and later). On ARM-based systems Tracy will try to use the timer register (~40 ns resolution). If it fails (due to kernel configuration), Tracy falls back to system provided timer, which can range in resolution from 250 ns to 1 µs.

system, which do have significantly reduced resolution (about 300 ns – 1 µs). This is enough to hide the subtle impact of cache access optimization, etc.

1.2.1 Timer accuracy

You may wonder why it is important to have a truly high resolution timer[3]. After all, you only want to profile functions that have long execution times, and not some short-lived procedures, that have no impact on the application's run time.

It is wrong to think so. Optimizing a function to execute in 430 ns, instead of 535 ns (note that there is only a 100 ns difference) results in 14 ms savings if the function is executed 18000 times[4]. It may not seem like a big number, but this is how much time there is to render a complete frame in a 60 FPS game. Imagine that this is your particle processing loop.

You also need to understand how timer precision is reflected in measurement errors. Take a look at figure 1. There you can see three discrete timer tick events, which increase the value reported by the timer by 300 ns. You can also see four readings of time ranges, marked A_1, A_2; B_1, B_2; C_1, C_2 and D_1, D_2.

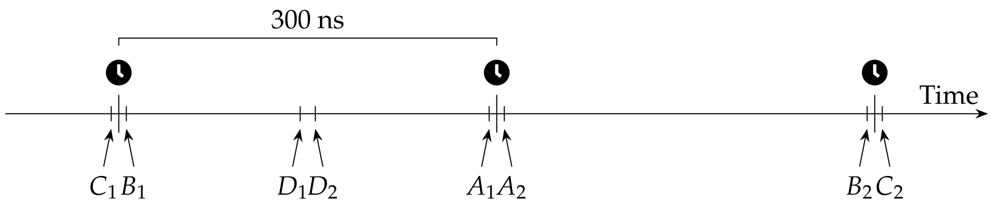

Figure 1: *Low precision (300 ns) timer. Discrete timer ticks are indicated by the icon.*

Now let's take a look at the timer readings.

- The A and D ranges both take a very short amount of time (10 ns), but the A range is reported as 300 ns, and the D range is reported as 0 ns.
- The B range takes a considerable amount of time (590 ns), but according to the timer readings, it took the same time (300 ns) as the short lived A range.
- The C range (610 ns) is only 20 ns longer than the B range, but it is reported as 900 ns, a 600 ns difference!

Here you can see why it is important to use a high precision timer. While there is no escape from the measurement errors, their impact can be reduced by increasing the timer accuracy.

1.3 Frame profiler

Tracy is aimed at understanding the inner workings of a tight loop of a game (or any other kind of an interactive application). That's why it slices the execution time of a program using the *frame*[5] as a basic work-unit[6]. The most interesting frames are the ones that took longer than the allocated time, producing visible hitches in the on-screen animation. Tracy allows inspection of such misbehavior.

[3]Interestingly, the `std::chrono::high_resolution_clock` is not really a high resolution clock.

[4]This is a real optimization case. The values are median function run times and do not reflect the real execution time, which explains the discrepancy in the total reported time.

[5]A frame is used to describe a single image displayed on the screen by the game (or any other program), preferably 60 times per second to achieve smooth animation. You can also think about physics update frames, audio processing frames, etc.

[6]Frame usage is not required. See section 3.3 for more information.

1.4 Sampling profiler

Tracy is able to periodically sample what the profiled application is doing, which gives you detailed performance information at the source line/assembly instruction level. This can give you deep understanding of how the program is executed by the processor. Using this information you can get a coarse view at the call stacks, fine-tune your algorithms, or even 'steal' an optimization performed by one compiler and make it available for the others.

1.5 Remote or embedded telemetry

Tracy uses the client-server model to enable a wide range of use-cases (see figure 2). For example, a game on a mobile phone may be profiled over the wireless connection, with the profiler running on a desktop computer. Or you can run the client and server on the same machine, using a localhost connection. It is also possible to embed the visualization front-end in the profiled application, making the profiling self-contained[7].

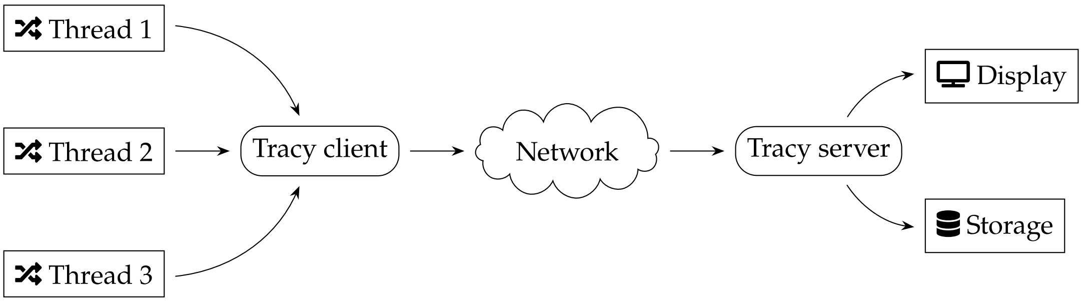

Figure 2: *Client-server model.*

In Tracy terminology, the profiled application is a *client* and the profiler itself is a *server*. It was named this way because the client is a thin layer that just collects events and sends them for processing and long-term storage on the server. The fact that the server needs to connect to the client to begin the profiling session may be a bit confusing at first.

1.6 Why Tracy?

You may wonder, why should you use Tracy, when there are so many other profilers available. Here are some arguments:

- Tracy is free and open source (BSD license), while RAD Telemetry costs about $8000 per year.
- Tracy provides out-of-the-box Lua bindings. It has been successfully integrated with other native and interpreted languages (Rust, Arma scripting language) using the C API (see chapter 3.12 for reference).
- Tracy has a wide variety of profiling options. You can profile CPU, GPU, locks, memory allocations, context switches and more.
- Tracy is feature rich. Statistical information about zones, trace comparisons, or inclusion of inline function frames in call stacks (even in statistics of sampled stacks) are features unique to Tracy.
- Tracy focuses on performance. Many tricks are used to reduce memory requirements and network bandwidth. The impact on the client execution speed is minimal, while other profilers perform heavy data processing within the profiled application (and then claim to be lightweight).
- Tracy uses low-level kernel APIs, or even raw assembly, where other profilers rely on layers of abstraction.

[7]See section 2.3.2 for guidelines.

- Tracy is multi-platform right from the very beginning. Both on the client and server side. Other profilers tend to have Windows-specific graphical interfaces.
- Tracy can handle millions of frames, zones, memory events, and so on, while other profilers tend to target very short captures.
- Tracy doesn't require manual markup of interesting areas in your code to start profiling. You may rely on automated call stack sampling and add instrumentation later, when you know where it's needed.
- Tracy provides mapping of source code to the assembly, with detailed information about cost of executing each instruction on the CPU.

1.7 Performance impact

To check how much slowdown is introduced by using Tracy, let's profile an example application. For this purpose we have used etcpak[8]. The input data was a 16384×16384 pixels test image and the 4×4 pixel block compression function was selected to be instrumented. The image was compressed on 12 parallel threads, and the timing data represents a mean compression time of a single image.

The results are presented in table 1. Dividing the average of run time differences (37.7 ms) by a number of captured zones per single image (16,777,216) shows us that the impact of profiling is only 2.25 ns per zone (this includes two events: start and end of a zone).

Mode	Zones (total)	Zones (single image)	Clean run	Profiling run	Difference
ETC1	201,326,592	16,777,216	110.9 ms	148.2 ms	+37.3 ms
ETC2	201,326,592	16,777,216	212.4 ms	250.5 ms	+38.1 ms

Table 1: *Zone capture time cost.*

1.7.1 Assembly analysis

To see how such small overhead (only 2.25 ns) is achieved, let's take a look at the assembly. The following x64 code is responsible for logging start of a zone. Do note that it is generated by compiling fully portable C++.

```
mov     byte ptr [rsp+0C0h],1           ; store zone activity information
mov     r15d,28h
mov     rax,qword ptr gs:[58h]          ; TLS
mov     r14,qword ptr [rax]             ; queue address
mov     rdi,qword ptr [r15+r14]         ; data address
mov     rbp,qword ptr [rdi+28h]         ; buffer counter
mov     rbx,rbp
and     ebx,7Fh                         ; 128 item buffer
jne     function+54h  -----------+      ; check if current buffer is usable
mov     rdx,rbp                  |
mov     rcx,rdi                  |
call    enqueue_begin_alloc      |      ; reclaim/alloc next buffer
shl     rbx,5  <-----------------+      ; buffer items are 32 bytes
add     rbx,qword ptr [rdi+48h]         ; calculate queue item address
mov     byte ptr [rbx],10h              ; queue item type
rdtsc                                   ; retrieve time
shl     rdx,20h
or      rax,rdx                         ; construct 64 bit timestamp
mov     qword ptr [rbx+1],rax           ; write timestamp
lea     rax,[__tracy_source_location]   ; static struct address
mov     qword ptr [rbx+9],rax           ; write source location data
lea     rax,[rbp+1]                     ; increment buffer counter
mov     qword ptr [rdi+28h],rax         ; write buffer counter
```

[8] https://github.com/wolfpld/etcpak

The second code block, responsible for ending a zone, is similar, but smaller, as it can reuse some variables retrieved in the above code.

1.8 Examples

To see how Tracy can be integrated into an application, you may look at example programs in the `examples` directory. Looking at the commit history might be the best way to do that.

1.9 On the web

Tracy can be found at the following web addresses:

- Homepage – `https://github.com/wolfpld/tracy`
- Bug tracker – `https://github.com/wolfpld/tracy/issues`
- Discord chat – `https://discord.gg/pk78auc`
- Sponsoring development – `https://github.com/sponsors/wolfpld/`

You may be also interested in a Bitbucket mirror: `https://bitbucket.org/wolfpld/tracy`.

2 First steps

Tracy Profiler supports MSVC, gcc and clang. A reasonably recent version of the compiler is needed, due to C++11 requirement. The following platforms are confirmed to be working (this is not a complete list):

- Windows (x86, x64)
- Linux (x86, x64, ARM, ARM64)
- Android (ARM, ARM64, x86)
- FreeBSD (x64)
- Cygwin (x64)
- MinGW (x64)
- WSL (x64)
- OSX (x64)
- iOS (ARM, ARM64)

Moreover, the following platforms are not supported due to how secretive their owners are, but were reported to be working after extending the system integration layer:

- PlayStation 4
- Xbox One
- Nintendo Switch
- Google Stadia

2.1 Initial client setup

The recommended way to integrate Tracy into an application is to create a git submodule in the repository (assuming that git is used for version control). This way it is very easy to update Tracy to newly released versions. If that's not an option, copy all files from the Tracy checkout directory to your project.

> **What revision should I use?**
>
> When deciding on the Tracy Profiler version you want to use, you have basically two options. Take into consideration the following pros and cons:
>
> - Using the last-version-tagged revision will give you a stable platform to work with. You won't experience any breakages, major UI overhauls or network protocol changes. Unfortunately, you also won't be getting any bug fixes.
> - Working with the bleeding edge `master` development branch will give you access to all the new improvements and features added to the profiler. While it is generally expected that `master` should always be usable, **there are no guarantees that it will be so.**
>
> Do note that all bug fixes and pull requests are made against the `master` branch.

With the source code included in your project, add the `tracy/TracyClient.cpp` source file to the IDE project and/or makefile. You're done. Tracy is now integrated into the application.

In the default configuration Tracy is disabled. This way you don't have to worry that the production builds will perform collection of profiling data. You will probably want to create a separate build configuration, with the `TRACY_ENABLE` define, which enables profiling. Be careful to enter the define name as specified, don't make a mistake of adding an additional `D` at the end. Also make sure that this macro is defined for all files across your project.

The application you want to profile should be compiled with all the usual optimization options enabled (i.e. make a release build). It makes no sense to profile debugging builds, as the unoptimized code and additional checks (asserts, etc.) completely change how the program behaves.

Finally, on Unix make sure that the application is linked with libraries `libpthread` and `libdl`. BSD systems will also need to be linked with `libexecinfo`.

2.1.1 Short-lived applications

In case you want to profile a short-lived program (for example, a compression utility that finishes its work in one second), set the `TRACY_NO_EXIT` environment variable to 1. With this option enabled, Tracy will not exit until an incoming connection is made, even if the application has already finished executing. If your platform doesn't support easy setup of environment variables, you may also add the `TRACY_NO_EXIT` define to your build configuration, which has the same effect.

2.1.2 On-demand profiling

By default Tracy will begin profiling even before the program enters the `main` function. If you don't want to perform a full capture of application life-time, you may define the `TRACY_ON_DEMAND` macro, which will enable profiling only when there's an established connection with the server.

It should be noted, that if on-demand profiling is *disabled* (which is the default), then the recorded events will be stored in the system memory until a server connection is made and the data can be uploaded[9]. Depending on the amount of the things profiled, the requirements for event storage can easily grow up to a couple of gigabytes. Since this data is cleared after the initial connection is made, you won't be able to perform a second connection to a client, unless the on-demand mode is used.

[9] This memory is never released, but it is reused for collection of further events.

Caveats

The client with on-demand profiling enabled needs to perform additional bookkeeping, in order to present a coherent application state to the profiler. This incurs additional time cost for each profiling event.

2.1.3 Client discovery

By default Tracy client will announce its presence to the local network[10]. If you want to disable this feature, define the `TRACY_NO_BROADCAST` macro.

2.1.4 Client network interface

By default Tracy client will listen on all network interfaces. If you want to restrict it to only listening on the localhost interface, define the `TRACY_ONLY_LOCALHOST` macro.

2.1.5 Setup for multi-DLL projects

In projects that consist of multiple DLLs/shared objects things are a bit different. Compiling `TracyClient.cpp` into every DLL is not an option because this would result in several instances of Tracy objects lying around in the process. We rather need to pass the instances of them to the different DLLs to be reused there.

For that you need a *profiler DLL* to which your executable and the other DLLs link. If that doesn't exist you have to create one explicitly for Tracy[11]. This library should contain the `tracy/TracyClient.cpp` source file. Link the executable and all DLLs which you want to profile to this DLL.

If you are targeting Windows with Microsoft Visual Studio or MinGW, add the `TRACY_IMPORTS` define to your application.

2.1.6 Problematic platforms

Some OS vendors think that *they* own and control the devices *you* have paid for. This results in restricting usage of APIs that might 'confuse' you, or denying you access to information about what your computer is doing.

This is a very sad state of things.

2.1.6.1 Apple woes

Because Apple *has* to be *think different*, there are some problems with using Tracy on OSX and iOS. First, the performance hit due to profiling is higher than on other platforms. Second, some critical features are missing and won't be possible to achieve:

- There's no support for the `TRACY_NO_EXIT` mode.
- Profiling is interrupted when the application exits. This will result in missing zones, memory allocations, or even source location names.
- OpenGL can't be profiled.

[10] Additional configuration may be required to achieve full functionality, depending on your network layout. Read about UDP broadcasts for more information.

[11] You may also look at the `library` directory in the profiler source tree.

2.1.6.2 Android lunacy

Starting with Android 8.0 you are no longer allowed to use the `/proc` file system. One of the consequences of this change is inability to check system CPU usage.

This is apparently a security enhancement. In its infinite wisdom Google has decided to not give you any option to bypass this restriction.

To workaround this limitation, you will need to have a rooted device. Execute the following commands using `root` shell:

```
setenforce 0
mount -o remount,hidepid=0 /proc
```

The first command will allow access to system CPU statistics. The second one will allow inspection of foreign processes (which is required for context switch capture). *Be sure that you are fully aware of the consequences of making these changes.*

2.1.6.3 Cloud service providers

In some cases you actually don't own the hardware, but lend it from someone else. In such circumstances you might be running inside a virtual machine, which may be configured to prohibit you from using the bare metal facilities needed by Tracy[12]. One example of such limitation would be lack of access to a reliable time stamp register readings, which will prevent the application from starting with either 'CPU doesn't support RDTSC instruction' or 'CPU doesn't support invariant TSC' error message. If you are using Windows, you may workaround this issue by rebuilding the profiled application with the `TRACY_TIMER_QPC` macro, but be aware that it will severely lower the resolution of timer readings.

2.1.7 Changing network port

Network communication between the client and the server by default is performed using network port 8086. The profiling session utilizes the TCP protocol and client broadcasts are done over UDP.

If for some reason you want to use another port[13], you can change it using the `TRACY_DATA_PORT` macro for the data connection, and `TRACY_BROADCAST_PORT` macro for client broadcasts. Alternatively, both ports may be changed at the same time by declaring the `TRACY_PORT` macro (specific macros listed before have higher priority). The data connection port may be also changed without recompiling the client application, by setting the `TRACY_PORT` environment variable.

If a custom port is not specified and the default listening port is already occupied, the profiler will automatically try to listen on a number of other ports.

Important

To enable network communication, Tracy needs to open a listening port. Make sure it is not blocked by an overzealous firewall or anti-virus program.

2.1.8 Limitations

When using Tracy Profiler, keep in mind the following requirements:

- Each lock may be used in no more than 64 unique threads.

[12] Or you might just be using a quite old CPU, which doesn't have support for required features.

[13] For example, other programs may already be using it, or you may have overzealous firewall rules, or you may want to run two clients on the same IP address.

- There can be no more than 65534 unique source locations[14]. This number is further split in half between native code source locations and dynamic source locations (for example, when Lua instrumentation is used).
- Profiling session cannot be longer than 1.6 days (2^{47} ns). This also includes on-demand sessions.
- No more than 4 billion (2^{32}) memory free events may be recorded.
- No more than 16 million (2^{24}) unique call stacks can be captured.

The following conditions also need apply, but don't trouble yourself with them too much. You would probably already knew, if you'd be breaking any.

- Only little-endian CPUs are supported.
- Virtual address space must be limited to 48 bits.
- Tracy server requires CPU which is able to handle misaligned memory accesses.

2.2 Check your environment

It is not an easy task to reliably measure performance of an application on modern machines. There are many factors affecting program execution characteristics, some of which you will be able to minimize, and others you will have to live with. It is critically important that you understand how these variables impact profiling results, as it is key to understanding the data you get.

2.2.1 Operating system

In a multitasking operating system applications compete for system resources with each other. This has a visible effect on the measurements performed by the profiler, which you may, or may not accept.

In order to get the most accurate profiling results you should minimize interference caused by other programs running on the same machine. Before starting a profile session close all web browsers, music players, instant messengers, and all other non-essential applications like Steam, Uplay, etc. Make sure you don't have the debugger hooked into the profiled program, as it also has impact on the timing results.

Interference caused by other programs can be seen in the profiler, if context switch capture (section 3.13.2) is enabled.

> **Debugger in Visual Studio**
>
> In MSVC you would typically run your program using the *Start Debugging* menu option, which is conveniently available as a F5 shortcut. You should instead use the *Start Without Debugging* option, available as Ctrl + F5 shortcut.

2.2.2 CPU design

Where to even begin here? Modern processors are such a complex beasts, that it's almost impossible to surely say anything about how they will behave. Cache configuration, prefetcher logic, memory timings, branch predictor, execution unit counts are all the drivers of instructions-per-cycle uplift nowadays, after the megahertz race had hit the wall. Not only is it incredibly difficult to reason about, but you also need to take into account how the CPU topology affects things, which is described in more detail in section 3.13.3.

Nevertheless, let's take a look on the ways we can try to stabilize the profiling data.

[14]A source location is a place in the code, which is identified by source file name and line number, for example when you markup a zone.

2.2.2.1 Superscalar out-of-order speculative execution

Also known as: the *spectre* thing we have to dealt with now.

You must be aware that most processors available on the market[15] *do not* execute machine code in a linear way, as laid out in the source code. This can lead to counterintuivive timing results reported by Tracy. Trying to get more 'reliable' readings[16] would require a change in the behavior of the code and this is not a thing a profiler should do. Instead, Tracy shows you what the hardware is *really* doing.

This is a complex subject and the details vary from one CPU to another. You can read a brief rundown of the topic at the following address: `https://travisdowns.github.io/blog/2019/06/11/speed-limits.html`.

2.2.2.2 Simultaneous multithreading

Also known as: Hyper-threading. Typically present on Intel and AMD processors.

To get the most reliable results you should have whole CPU core resources dedicated to a single thread of your program. Otherwise you're no longer measuring the behavior of your code, but rather how it keeps up when its computing resources are randomly taken away by some other thing running on another pipeline within the same physical core.

Note that you might *want* to observe this behavior, if you plan to deploy your application on a machine with simultaneous multithreading enabled. This would require careful examination of what else is running on the machine, or even how the threads of your own program are scheduled by the operating system, as various combinations of competing workloads (e.g. integer/floating point operations) will be impacted differently.

2.2.2.3 Turbo mode frequency scaling

Also known as: Turbo Boost (Intel), Precision Boost (AMD).

While the CPU is more-or-less designed to always be able to work at the advertised *base* frequency, there is usually some headroom left, which allows usage of the built-in automatic overclocking. There are no guarantees that the turbo frequencies can be attained, or how long they will be held, as there are many things to take into consideration:

- How many cores are being used? Just one, or all 8? All 16?
- What type of work is being performed? Integer? Floating point? 128-wide SIMD? 256-wide SIMD? 512-wide SIMD?
- Were you lucky in the silicon lottery? Some dies are simply better made and are able to achieve higher frequencies.
- Are you running on the best-rated core, or at the worst-rated core? Some cores may be unable to match the performance of other cores in the same processor.
- What kind of cooling solution are you using? The cheap one bundled with the CPU, or a beefy chunk of metal that has no problem with heat dissipation?
- Do you have complete control over the power profile? Spoiler alert: no. The operating system may run anything at any time on any of the other cores, which will impact the turbo frequency you're able to achieve.

As you can see, this feature basically screams 'unreliable results!' Best keep it disabled and run at the base frequency. Otherwise your timings won't make much sense. A true example: branchless compression function executing multiple times with the same input data was measured executing at *four* different speeds.

Keep in mind that even at the base frequency you may hit thermal limits of the silicon and be downthrottled.

[15] With the exception of low-cost ARM CPUs.

[16] And by saying 'reliable' you do in reality mean: behaving in a way you expect it to.

2.2.2.4 Power saving

This is basically the same as turbo mode, but in reverse. While unused, processor cores are kept at lower frequencies (or even completely disabled) to reduce power usage. When your code starts running[17] the core frequency needs to ramp up, which may be visible in the measurements.

What's even worse, if your code doesn't do a lot of work (for example, because it is waiting for the GPU to finish rendering the frame), the core frequency might not be ramped up to 100%, which will skew the results.

Again, to get the best results, keep this feature disabled.

2.2.2.5 AVX offset and power licenses

Intel CPUs are unable to run at their advertised frequencies when wide SIMD operations are performed due to increased power requirements[18]. Depending on the width *and* type of operations performed, the core operating frequency will be reduced, in some cases quite drastically[19]. To make things even better, *some* part of the workload will be executed within the available power license, at twice reduced processing rate, then the CPU may be stopped for some time, so that the wide parts of executions units may be powered up, then the work will continue at full processing rate, but at reduced frequency.

Be very careful when using AVX2 or AVX512.

More information can be found at `https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html`, `https://en.wikichip.org/wiki/intel/frequency_behavior`.

2.2.2.6 Summing it up

Power management schemes employed in various CPUs make it hard to reason about true performance of the code. For example, figure 3 contains a histogram of function execution times (as described in chapter 5.6), as measured on an AMD Ryzen CPU. The results ranged from 13.05 µs to 61.25 µs (extreme outliers were not included on the graph, limiting the longest displayed time to 36.04 µs).

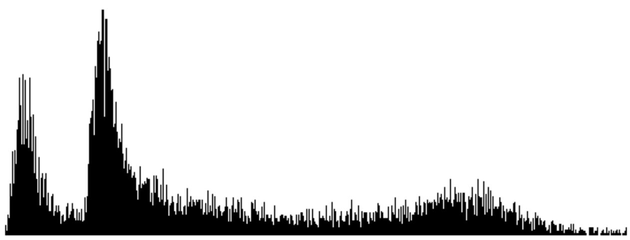

Figure 3: *Example function execution times on a Ryzen CPU*

We can immediately see that there are two distinct peaks, at 13.4 µs and 15.3 µs. A reasonable assumption would be that there are two paths in the code, one that can omit some work, and the second one which must do some additional job. But here's a catch – the measured code is actually branchless and is always executed the same way. The two peaks represent two turbo frequencies between which the CPU was aggressively switching.

We can also see that the graph gradually falls off to the right (representing longer times), with a small bump near the end. This can be attributed to running in power saving mode, with differing reaction times to the required operating frequency boost to full power.

[17] Not necessarily when the application is started, but also when, for example, a blocking mutex becomes released by other thread and is acquired.

[18] AMD processors are not affected by this issue.

[19] `https://en.wikichip.org/wiki/intel/xeon_gold/5120#Frequencies`

2.3 Building the server

The easiest way to get going is to build the data analyzer, available in the `profiler` directory. With it you can connect to localhost or remote clients and view the collected data right away.

If you prefer to inspect the data only after a trace has been performed, you may use the command line utility in the `capture` directory. It will save a data dump that may be later opened in the graphical viewer application.

Note that ideally you should be using the same version of the Tracy profiler on both client and server. The network protocol may change in between versions, in which case you won't be able to make a connection.

See section 4 for more information about performing captures.

> **Important**
>
> Due to the memory requirements for data storage, Tracy server is only supposed to run on 64-bit platforms. While there is nothing preventing the program from building and executing in a 32-bit environment, doing so is not supported.

2.3.1 Required libraries

To build the application contained in the `profiler` directory, you will need to install external libraries, which are not bundled with Tracy.

Capstone library At the time of writing, the capstone library is in a bit of disarray. The officially released version 4.0.2 can't disassemble some AVX instructions, which are successfully parsed by the `next` branch. However, the `next` branch somehow lost information about input/output registers for some functions. You may want to explore the various available versions to find one that suits your needs the best.

2.3.1.1 Windows

On Windows you will need to use the `vcpkg` utility. If you are not familiar with this tool, please read the description at the following address: `https://docs.microsoft.com/en-us/cpp/build/vcpkg`.

There are two ways you can run `vcpkg` to install the dependencies for Tracy:

- Local installation within the project directory – run this script to download and build both `vcpkg` and the required dependencies:

  ```
  vcpkg\install_vcpkg_dependencies.bat
  ```

  This writes files only to the `vcpkg\vcpkg` directory and makes no other changes on your machine.

- System-wide installation – install `vcpkg` by following the instructions on its website, and then execute the following commands:

  ```
  vcpkg integrate install
  vcpkg install --triplet x64-windows-static freetype glfw3 capstone[arm,arm64,x86]
  ```

2.3.1.2 Unix

On Unix systems you will need to install the `pkg-config` utility and the following libraries: `glfw`, `freetype`, `capstone`. Some Linux distributions will require you to add a `lib` prefix and a `-dev`, or `-devel` postfix to library names. You may also need to add a seemingly random number to the library name (for example: `freetype2`, or `freetype6`). How fun!

Installation of the libraries on OSX can be facilitated using the `brew` package manager.

2.3.2 Embedding the server in profiled application

While not officially supported, it is possible to embed the server in your application, the same one which is running the client part of Tracy. This is left up for you to figure out.

Note that most libraries bundled with Tracy are modified in some way and contained in the `tracy` namespace. The one exception is Dear ImGui, which can be freely replaced.

Be aware that while the Tracy client uses its own separate memory allocator, the server part of Tracy will use global memory allocation facilities, shared with the rest of your application. This will affect both the memory usage statistics and Tracy memory profiling.

The following defines may be of interest:

- `TRACY_NO_FILESELECTOR` – controls whether a system load/save dialog is compiled in. If it's enabled, the saved traces will be named `trace.tracy`.
- `TRACY_NO_STATISTICS` – Tracy will perform statistical data collection on the fly, if this macro is *not* defined. This allows extended analysis of the trace (for example, you can perform a live search for matching zones) at a small CPU processing cost and a considerable memory usage increase (at least 8 bytes per zone).
- `TRACY_NO_ROOT_WINDOW` – the main profiler view won't occupy whole window if this macro is defined. Additional setup is required for this to work. If you are embedding the server into your application you probably want to enable this option.

2.4 Naming threads

Remember to set thread names for proper identification of threads. You must use the functions exposed in the `tracy/common/TracySystem.hpp` header to do so, as the system facilities typically have limited functionality.

If context switch capture is active, Tracy will try to capture thread names through operating system data. This is only a fallback mechanism and it shouldn't be relied upon.

2.5 Crash handling

On selected platforms (see section 2.6) Tracy will intercept application crashes[20]. This serves two purposes. First, the client application will be able to send the remaining profiling data to the server. Second, the server will receive a crash report with information about the crash reason, call stack at the time of crash, etc.

This is an automatic process and it doesn't require user interaction.

Caveats

> On MSVC the debugger has priority over the application in handling exceptions. If you want to finish the profiler data collection with the debugger hooked-up, select the *continue* option in the debugger pop-up dialog.

2.6 Feature support matrix

Some features of the profiler are only available on selected platforms. Please refer to table 2 for details.

[20]For example, invalid memory accesses ('segmentation faults', 'null pointer exceptions'), divisions by zero, etc.

Feature	Windows	Linux	Android	OSX	iOS	BSD
Profiling initialization	✔	✔	✔	💩	💩	✔
CPU zones	✔	✔	✔	✔	✔	✔
Locks	✔	✔	✔	✔	✔	✔
Plots	✔	✔	✔	✔	✔	✔
Messages	✔	✔	✔	✔	✔	✔
Memory	✔	✔	✔	✔	✔	✔
GPU zones (OpenGL)	✔	✔	✔	💩	💩	
GPU zones (Vulkan)	✔	✔	✔	✔	✔	
Call stacks	✔	✔	✔	✔	✔	✔
Symbol resolution	✔	✔	✔	✔	✔	✔
Crash handling	✔	✔	✔	✘	✘	✘
CPU usage probing	✔	✔	✔	✔	✔	✔
Context switches	✔	✔	✔	✘	💩	✘
CPU topology information	✔	✔	✔	✘	✘	✘
Call stack sampling	✔	✘	✘	✘	💩	✘

💩 – Not possible to support due to platform limitations.

Table 2: *Feature support matrix*

3 Client markup

With the aforementioned steps you will be able to connect to the profiled program, but there probably won't be any data collection performed[21]. Unless you're able to perform automatic call stack sampling (see chapter 3.13.4), you will have to manually instrument the application. All the user-facing interface is contained in the `tracy/Tracy.hpp` header file.

Manual instrumentation is best started with adding markup to the main loop of the application, along with a few function that are called there. This will give you a rough outline of the function's time cost, which you may then further refine by instrumenting functions deeper in the call stack. Alternatively, automated sampling might guide you more quickly to places of interest.

3.1 Handling text strings

When dealing with Tracy macros, you will encounter two ways of providing string data to the profiler. In both cases you should pass `const char*` pointers, but there are differences in expected life-time of the pointed data.

1. When a macro only accepts a pointer (for example: `TracyMessageL(text)`), the provided string data must be accessible at any time in program execution (*this also includes the time after exiting the `main` function*). The string also cannot be changed. This basically means that the only option is to use a string literal (e.g.: `TracyMessageL("Hello")`).

2. If there's a string pointer with a size parameter (for example: `TracyMessage(text, size)`), the profiler will allocate an internal temporary buffer to store the data. The pointed-to data is not used afterwards. You should be aware that allocating and copying memory involved in this operation has a small time cost.

[21]With some small exceptions, see section 3.13.

3.2 Specifying colors

In some cases you will want to provide your own colors to be displayed by the profiler. In all such places you should use a hexadecimal `0xRRGGBB` notation.

Alternatively you may use named colors predefined in `common/TracyColor.hpp` (included by `Tracy.hpp`). Visual reference: `https://en.wikipedia.org/wiki/X11_color_names`.

Do not use `0x000000` if you want to specify black color, as zero is a special value indicating that no color was set. Instead, use a value close to zero, e.g. `0x000001`.

3.3 Marking frames

To slice the program's execution recording into frame-sized chunks[22], put the `FrameMark` macro after you have completed rendering the frame. Ideally that would be right after the swap buffers command.

Do I need this?

This step is optional, as some applications do not use the concept of a frame.

3.3.1 Secondary frame sets

In some cases you may want to track more than one set of frames in your program. To do so, you may use the `FrameMarkNamed(name)` macro, which will create a new set of frames for each unique name you provide.

3.3.2 Discontinuous frames

Some types of frames are discontinuous by nature. For example, a physics processing step in a game loop, or an audio callback running on a separate thread. These kinds of workloads are executed periodically, with a pause between each run. Tracy can also track these kind of frames.

To mark the beginning of a discontinuous frame use the `FrameMarkStart(name)` macro. After the work is finished, use the `FrameMarkEnd(name)` macro.

Important

- Frame types *must not* be mixed. For each frame set, identified by an unique name, use either continuous or discontinuous frames only!
- You *must* issue the `FrameMarkStart` and `FrameMarkEnd` macros in proper order. Be extra careful, especially if multi-threading is involved.
- Discontinuous frames may not work correctly if the profiled program doesn't have string pooling enabled. This is an implementation issue which will be fixed in the future.

3.3.3 Frame images

It is possible to attach a screen capture of your application to any frame in the main frame set. This can help you see the context of what's happening in various places in the trace. You need to implement retrieval of the image data from GPU by yourself.

Images are sent using the `FrameImage(image, width, height, offset, flip)` macro, where `image` is a pointer to RGBA[23] pixel data, `width` and `height` are the image dimensions, which *must be divisible by 4*,

[22] Each frame starts immediately after the previous has ended.
[23] Alpha value is ignored, but leaving it out wouldn't map well to the way graphics hardware works.

`offset` specifies how much frame lag was there for the current image (see chapter 3.3.3.1), and `flip` should be set, if the graphics API stores images upside-down[24]. The image data is copied by the profiler and doesn't need to be retained.

Handling image data requires a lot of memory and bandwidth[25]. To achieve sane memory usage you should scale down taken screen shots to a sensible size, e.g. 320 × 180.

To further reduce image data size, frame images are internally compressed using the DXT1 Texture Compression technique[26], which significantly reduces data size[27], at a small quality decrease. The compression algorithm is very fast and can be made even faster by enabling SIMD processing, as indicated in table 3.

Implementation	Required define	Time
x86 Reference	—	198.2 µs
x86 SSE4.1[a]	`__SSE4_1__`	25.4 µs
x86 AVX2	`__AVX2__`	17.4 µs
ARM Reference	—	1.04 ms
ARM32 NEON[b]	`__ARM_NEON`	529 µs
ARM64 NEON	`__ARM_NEON`	438 µs

[a)] VEX encoding; [b)] ARM32 NEON code compiled for ARM64

Table 3: *Client compression time of* 320 × 180 *image. x86: Ryzen 9 3900X (MSVC); ARM: ODROID-C2 (gcc).*

Caveats

- Frame images are compressed on a second client profiler thread[a], to reduce memory usage of queued images. This might have impact on the performance of the profiled application.
- Due to implementation details of the network buffer, single frame image cannot be greater than 256 KB after compression. Note that a 960 × 540 image fits in this limit.

[a]Small part of compression task is performed on the server.

3.3.3.1 OpenGL screen capture code example

There are many pitfalls associated with retrieving screen contents in an efficient way. For example, using `glReadPixels` and then resizing the image using some library is terrible for performance, as it forces synchronization of the GPU to CPU and performs the downscaling in software. To do things properly we need to scale the image using the graphics hardware and transfer data asynchronously, which allows the GPU to run independently of CPU.

The following example shows how this can be achieved using OpenGL 3.2. More recent OpenGL versions allow doing things even better (for example by using persistent buffer mapping), but it won't be covered here.

Let's begin by defining the required objects. We need a *texture* to store the resized image, a *framebuffer object* to be able to write to the texture, a *pixel buffer object* to store the image data for access by the CPU and a *fence* to know when the data is ready for retrieval. We need everything in *at least* three copies (we'll use four), because the rendering, as seen in program, may be ahead of the GPU by a couple frames. We need an index to access the appropriate data set in a ring-buffer manner. And finally, we need a queue to store indices to data sets that we are still waiting for.

```
GLuint m_fiTexture[4];
```

[24]For example, OpenGL flips images, but Vulkan does not.
[25]One uncompressed 1080p image takes 8 MB.
[26]`https://en.wikipedia.org/wiki/S3_Texture_Compression`
[27]One pixel is stored in a nibble (4 bits) instead of 32 bits.

```
GLuint m_fiFramebuffer[4];
GLuint m_fiPbo[4];
GLsync m_fiFence[4];
int m_fiIdx = 0;
std::vector<int> m_fiQueue;
```

Everything needs to be properly initialized (the cleanup is left for the reader to figure out).

```
glGenTextures(4, m_fiTexture);
glGenFramebuffers(4, m_fiFramebuffer);
glGenBuffers(4, m_fiPbo);
for(int i=0; i<4; i++)
{
    glBindTexture(GL_TEXTURE_2D, m_fiTexture[i]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 320, 180, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);

    glBindFramebuffer(GL_FRAMEBUFFER, m_fiFramebuffer[i]);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
        m_fiTexture[i], 0);

    glBindBuffer(GL_PIXEL_PACK_BUFFER, m_fiPbo[i]);
    glBufferData(GL_PIXEL_PACK_BUFFER, 320*180*4, nullptr, GL_STREAM_READ);
}
```

We will now setup a screen capture, which will downscale the screen contents to 320×180 pixels and copy the resulting image to a buffer which will be accessible by the CPU when the operation is done. This should be placed right before *swap buffers* or *present* call.

```
assert(m_fiQueue.empty() || m_fiQueue.front() != m_fiIdx);  // check for buffer overrun
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, m_fiFramebuffer[m_fiIdx]);
glBlitFramebuffer(0, 0, res.x, res.y, 0, 0, 320, 180, GL_COLOR_BUFFER_BIT, GL_LINEAR);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBindFramebuffer(GL_READ_FRAMEBUFFER, m_fiFramebuffer[m_fiIdx]);
glBindBuffer(GL_PIXEL_PACK_BUFFER, m_fiPbo[m_fiIdx]);
glReadPixels(0, 0, 320, 180, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);
glBindFramebuffer(GL_READ_FRAMEBUFFER, 0);
m_fiFence[m_fiIdx] = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
m_fiQueue.emplace_back(m_fiIdx);
m_fiIdx = (m_fiIdx + 1) % 4;
```

And lastly, just before the capture setup code that was just added[28] we need to have the image retrieval code. We are checking if the capture operation has finished and if it has, we map the *pixel buffer object* to memory, inform the profiler that there's image data to be handled, unmap the buffer and go to check the next queue item. If a capture is still pending, we break out of the loop and wait until the next frame to check if the GPU has finished the capture.

```
while(!m_fiQueue.empty())
{
    const auto fiIdx = m_fiQueue.front();
    if(glClientWaitSync(m_fiFence[fiIdx], 0, 0) == GL_TIMEOUT_EXPIRED) break;
    glDeleteSync(m_fiFence[fiIdx]);
    glBindBuffer(GL_PIXEL_PACK_BUFFER, m_fiPbo[fiIdx]);
    auto ptr = glMapBufferRange(GL_PIXEL_PACK_BUFFER, 0, 320*180*4, GL_MAP_READ_BIT);
    FrameImage(ptr, 320, 180, m_fiQueue.size());
    glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
    m_fiQueue.erase(m_fiQueue.begin());
}
```

[28] Yes, before. We are handling past screen captures here.

Notice that in the call to `FrameImage` we are passing the remaining queue size as the `offset` parameter. Queue size represents how many frames ahead our program is relative to the GPU. Since we are sending past frame images we need to specify how many frames behind the images are. Of course if this would be a synchronous capture (without use of fences and with retrieval code after the capture setup), we would set `offset` to zero, as there would be no frame lag.

High quality capture The code above uses `glBlitFramebuffer` function, which can only use nearest neighbor filtering. This can result in low-quality screen shots, as shown on figure 4. With a bit more work it is possible to obtain much nicer looking screen shots, as presented on figure 5. Unfortunately, you will need to setup a complete rendering pipeline for this to work.

First, you need to allocate additional set of intermediate frame buffers and textures, sized the same as the screen. These new textures should have minification filter set to `GL_LINEAR_MIPMAP_LINEAR`. You will also need to setup everything needed to render a full-screen quad: a simple texturing shader and vertex buffer with appropriate data. Since this vertex buffer will be used to render to the scaled-down framebuffer, you may prepare its contents beforehand and update it only when the aspect ratio would change.

With all this done, the screen capture can be performed as follows:

- Setup vertex buffer configuration for the full-screen quad buffer (you only need position and uv coordinates).
- Blit the screen contents to the full-sized framebuffer.
- Bind the texture backing the full-sized framebuffer.
- Generate mip-maps using `glGenerateMipmap`.
- Set viewport to represent the scaled-down image size.
- Bind vertex buffer data, shader, setup the required uniforms.
- Draw full-screen quad to the scaled-down framebuffer.
- Retrieve framebuffer contents, as in the code above.
- Restore viewport, vertex buffer configuration, bound textures, etc.

While this approach is much more complex than the previously discussed one, the resulting image quality increase makes it worth it.

Figure 4: *Low-quality screen shot*

Figure 5: *High-quality screen shot*

You can see the performance results you may expect in a simple application in table 4. The naïve capture performs synchronous retrieval of full screen image and resizes it using *stb_image_resize*. The proper and high quality captures do things as described in this chapter.

Resolution	Naïve capture	Proper capture	High quality
1280 × 720	80 FPS	4200 FPS	2800 FPS
2560 × 1440	23 FPS	3300 FPS	1600 FPS

Table 4: *Frame capture efficiency*

3.4 Marking zones

To record a zone's[29] execution time add the `ZoneScoped` macro at the beginning of the scope you want to measure. This will automatically record function name, source file name and location. Optionally you may use the `ZoneScopedC(color)` macro to set a custom color for the zone. Note that the color value will be constant in the recording (don't try to parametrize it). You may also set a custom name for the zone, using the `ZoneScopedN(name)` macro. Color and name may be combined by using the `ZoneScopedNC(name, color)` macro.

Use the `ZoneText(text, size)` macro to add a custom text string that will be displayed along the zone information (for example, name of the file you are opening). Multiple text strings can be attached to any single zone. If you want to send a numeric value and don't want to pay the cost of converting it to a string, you may use the `ZoneValue(uint64_t)` macro.

If you want to set zone name on a per-call basis, you may do so using the `ZoneName(text, size)` macro. This name won't be used in the process of grouping the zones for statistical purposes (sections 5.5 and 5.6).

3.4.1 Multiple zones in one scope

Using the `ZoneScoped` family of macros creates a stack variable named `___tracy_scoped_zone`. If you want to measure more than one zone in the same scope, you will need to use the `ZoneNamed` macros, which require that you provide a name for the created variable. For example, instead of `ZoneScopedN("Zone name")`, you would use `ZoneNamedN(variableName, "Zone name", true)`[30].

The `ZoneText`, `ZoneValue` and `ZoneName` macros apply to the zones created using the `ZoneScoped` macros. For zones created using the `ZoneNamed` macros, you can use the `ZoneTextV(variableName, text, size)`, `ZoneValueV(variableName, uint64_t)`, or `ZoneNameV(variableName, text, size)` macros, or invoke the methods `Text`, `Value` or `Name` directly on the variable you have created.

Zone stack

The `ZoneScoped` macros are imposing creation and usage of an implicit zone stack. You must follow the rules of this stack also when you are using the named macros, which give you some more leeway in doing things. For example, you can only set the text for the zone which is on top of the stack, as you only could do with the `ZoneText` macro. It doesn't matter that you can call the `Text` method of a non-top zone which is accessible through a variable. Take a look at the following code:

```
{
    ZoneNamed(Zone1, true);
    (a)
    {
        ZoneNamed(Zone2, true);
        (b)
    }
    (c)
}
```

[29]A zone represents the life-time of a special on-stack profiler variable. Typically it would exist for the duration of a whole scope of the profiled function, but you also can measure time spent in scopes of a for-loop, or an if-branch.
[30]The last parameter is explained in section 3.4.3.

> It is valid to set the `Zone1` text or name *only* in places a or c. After `Zone2` is created at b you can no longer perform operations on `Zone1`, until `Zone2` is destroyed.

3.4.2 Variable shadowing

The following code is fully compliant with the C++ standard:

```
void Function()
{
    ZoneScoped;
    ...
    for(int i=0; i<10; i++)
    {
        ZoneScoped;
        ...
    }
}
```

This doesn't stop some compilers from dispensing *fashion advice* about variable shadowing (as both `ZoneScoped` calls create a variable with the same name, with the inner scope one shadowing the one in the outer scope). If you want to avoid these warnings, you will also need to use the `ZoneNamed` macros.

3.4.3 Filtering zones

Zone logging can be disabled on a per zone basis, by making use of the `ZoneNamed` macros. Each of the macros takes an `active` argument ('`true`' in the example in section 3.4.1), which will determine whether the zone should be logged.

Note that this parameter may be a run-time variable, for example an user controlled switch to enable profiling of a specific part of code only when required.

If the condition is constant at compile-time, the resulting code will not contain a branch (the profiling code will either be always enabled, or won't be there at all). The following listing presents how profiling of specific application subsystems might be implemented:

```
enum SubSystems
{
    Sys_Physics     = 1 << 0,
    Sys_Rendering   = 1 << 1,
    Sys_NasalDemons = 1 << 2
}

...

// Preferably a define in the build system
#define SUBSYSTEMS Sys_Physics | Sys_NasalDemons

...

void Physics::Process()
{
    ZoneScopedN( __tracy, SUBSYSTEMS & Sys_Physics );       // always true, no runtime cost
    ...
}

void Graphics::Render()
{
    ZoneScopedN( __tracy, SUBSYSTEMS & Sys_Graphics );      // always false, no runtime cost
    ...
}
```

3.4.4 Manual management of zone scope

The zone markup macros automatically report when they end, through the RAII mechanism[31]. This is very helpful, but sometimes you may want to mark the zone start and end points yourself, for example if you want to have a zone that crosses the function's boundary. This can be achieved by using the C API, which is described in section 3.12.

3.4.5 Exiting program from within a zone

At the present time exiting the profiled application from inside a zone is not supported. When the client calls `exit()`, the profiler will wait for all zones to end, before a program can be truly terminated. If program execution stopped inside a zone, this will never happen, and the profiled application will seemingly hang up. At this point you will need to manually terminate the program (or simply disconnect the profiler server).

As a workaround, you may add a `try/catch` pair at the bottom of the function stack (for example in the `main()` function) and replace `exit()` calls with throwing a custom exception. When this exception is caught, you may call `exit()`, knowing that the application's data structures (including profiling zones) were properly cleaned up.

3.5 Marking locks

Modern programs must use multi-threading to achieve full performance capability of the CPU. Correct execution requires claiming exclusive access to data shared between threads. When many threads want to enter the same critical section at once, the application's multi-threaded performance advantage is nullified. To help solve this problem, Tracy can collect and display lock interactions in threads.

To mark a lock (mutex) for event reporting, use the `TracyLockable(type, varname)` macro. Note that the lock must implement the Mutex requirement[32] (i.e. there's no support for timed mutices). For a concrete example, you would replace the line

```
std::mutex m_lock;
```

with

```
TracyLockable(std::mutex, m_lock);
```

Alternatively, you may use `TracyLockableN(type, varname, description)` to provide a custom lock name at a global level, which will replace the automatically generated '`std::mutex m_lock`'-like name. You may also set a custom name for a specific instance of a lock, through the `LockableName(varname, name, size)` macro.

The standard `std::lock_guard` and `std::unique_lock` wrappers should use the `LockableBase(type)` macro for their template parameter (unless you're using C++17, with improved template argument deduction). For example:

```
std::lock_guard<LockableBase(std::mutex)> lock(m_lock);
```

To mark the location of a lock being held, use the `LockMark(varname)` macro, after you have obtained the lock. Note that the `varname` must be a lock variable (a reference is also valid). This step is optional.

Similarly, you can use `TracySharedLockable`, `TracySharedLockableN` and `SharedLockableBase` to mark locks implementing the SharedMutex requirement[33]. Note that while there's no support for timed mutices in Tracy, both `std::shared_mutex` and `std::shared_timed_mutex` may be used[34].

[31] `https://en.cppreference.com/w/cpp/language/raii`
[32] `https://en.cppreference.com/w/cpp/named_req/Mutex`
[33] `https://en.cppreference.com/w/cpp/named_req/SharedMutex`
[34] Since `std::shared_mutex` was added in C++17, using `std::shared_timed_mutex` is the only way to have shared mutex functionality in C++14.

Condition variables

The standard `std::condition_variable` is only able to accept `std::mutex` locks. To be able to use Tracy lock wrapper, use `std::condition_variable_any` instead.

Caveats

Due to limits of internal bookkeeping in the profiler, each lock may be used in no more than 64 unique threads. If you have many short lived temporary threads, consider using a thread pool to limit the numbers of created threads.

3.5.1 Custom locks

If using the `TracyLockable` or `TracySharedLockable` wrappers does not fit your needs, you may want to add a more fine-grained instrumentation to your code. Classes `LockableCtx` and `SharedLockableCtx` contained in the `TracyLock.hpp` header contain all the required functionality. Lock implementations in classes `Lockable` and `SharedLockable` show how to properly perform context handling.

3.6 Plotting data

Tracy is able to capture and draw numeric value changes over time. You may use it to analyze draw call counts, number of performed queries, etc. To report data, use the `TracyPlot(name, value)` macro.

To configure how plot values are presented by the profiler, you may use the `TracyPlotConfig(name, format)` macro, where `format` is one of the following options:

- `tracy::PlotFormatType::Number` – values will be displayed as plain numbers.
- `tracy::PlotFormatType::Memory` – treats the values as memory sizes. Will display kilobytes, megabytes, etc.
- `tracy::PlotFormatType::Percentage` – values will be displayed as percentage (with value 100 being equal to 100%).

3.7 Message log

Fast navigation in large data sets and correlating zones with what was happening in application may be difficult. To ease these issues Tracy provides a message log functionality. You can send messages (for example, your typical debug output) using the `TracyMessage(text, size)` macro. Alternatively, use `TracyMessageL(text)` for string literal messages.

If you want to include color coding of the messages (for example to make critical messages easily visible), you can use `TracyMessageC(text, size, color)` or `TracyMessageLC(text, color)` macros.

3.7.1 Application information

Tracy can collect additional information about the profiled application, which will be available in the trace description. This can include data such as the source repository revision, the environment in which application is running (dev/prod), etc.

Use the `TracyAppInfo(text, size)` macro to report the data.

3.8 Memory profiling

Tracy can monitor memory usage of your application. Knowledge about each performed memory allocation enables the following:

- Memory usage graph (like in massif, but fully interactive).
- List of active allocations at program exit (memory leaks).
- Visualization of memory map.
- Ability to rewind view of active allocations and memory map to any point of program execution.
- Information about memory statistics of each zone.
- Memory allocation hot-spot tree.

To mark memory events, use the `TracyAlloc(ptr, size)` and `TracyFree(ptr)` macros. Typically you would do that in overloads of `operator new` and `operator delete`, for example:

```
void* operator new(std::size_t count)
{
    auto ptr = malloc(count);
    TracyAlloc(ptr, count);
    return ptr;
}

void operator delete(void* ptr) noexcept
{
    TracyFree(ptr);
    free(ptr);
}
```

Important

Each tracked memory free event must also have a corresponding memory allocation event. Tracy will terminate the profiling session if this assumption is broken (see section 4.6). If you encounter this issue, you may want to check for:

- Mismatched `malloc`/`new` or `free`/`delete`.
- Double freeing the memory.
- Untracked allocations made in external libraries, that are freed in the application.
- Places where the memory is allocated, but profiling markup is added.

This requirement is relaxed in the on-demand mode (section 2.1.2), because the memory allocation event might have happened before the connection was made.

3.9 GPU profiling

Tracy provides bindings for profiling OpenGL, Vulkan, Direct3D 12 and OpenCL execution time on GPU.

Note that the CPU and GPU timers may be not synchronized. You can correct the resulting desynchronization in the profiler's options (section 5.3).

3.9.1 OpenGL

You will need to include the `tracy/TracyOpenGL.hpp` header file and declare each of your rendering contexts using the `TracyGpuContext` macro (typically you will only have one context). Tracy expects no more than one context per thread and no context migration.

To mark a GPU zone use the `TracyGpuZone(name)` macro, where `name` is a string literal name of the zone. Alternatively you may use `TracyGpuZoneC(name, color)` to specify zone color.

You also need to periodically collect the GPU events using the `TracyGpuCollect` macro. A good place to do it is after the swap buffers function call.

Caveats

- GPU profiling is not supported on OSX, iOS[a].
- Android devices do work, if GPU drivers are not broken. Disjoint events are not currently handled, so some readings may be a bit spotty.
- Nvidia drivers are unable to provide consistent timing results when two OpenGL contexts are used simultaneously.
- Calling the `TracyGpuCollect` macro is a fairly slow operation (couple µs).

[a]Because Apple is unable to implement standards properly.

3.9.2 Vulkan

Similarly, for Vulkan support you should include the `tracy/TracyVulkan.hpp` header file. Tracing Vulkan devices and queues is a bit more involved, and the Vulkan initialization macro `TracyVkContext(physdev, device, queue, cmdbuf)` returns an instance of `TracyVkCtx` object, which tracks an associated Vulkan queue. Cleanup is performed using the `TracyVkDestroy(ctx)` macro. You may create multiple Vulkan contexts.

The physical device, logical device, queue and command buffer must relate with each other. The queue must support graphics or compute operations. The command buffer must be in the initial state and be able to be reset. It will be rerecorded and submitted to the queue multiple times and it will be in the executable state on exit from the initialization function.

To mark a GPU zone use the `TracyVkZone(ctx, cmdbuf, name)` macro, where `name` is a string literal name of the zone. Alternatively you may use `TracyVkZoneC(ctx, cmdbuf, name, color)` to specify zone color. The provided command buffer must be in the recording state and it must be created within the queue that is associated with `ctx` context.

You also need to periodically collect the GPU events using the `TracyVkCollect(ctx, cmdbuf)` macro[35]. The provided command buffer must be in the recording state and outside of a render pass instance.

3.9.3 Direct3D 12

To enable Direct3D 12 support, include the `tracy/TracyD3D12.hpp` header file. Tracing Direct3D 12 queues is nearly on par with the Vulkan implementation, where a `TracyD3D12Ctx` is returned from a call to `TracyD3D12Context(device, queue)`, which should be later cleaned up with the `TracyD3D12Destroy(ctx)` macro. Multiple contexts can be created, each with any queue type.

The queue must have been created through the specified device, however a command list is not needed for this stage.

Using GPU zones is the same as the Vulkan implementation, where the `TracyD3D12Zone(ctx, cmdList, name)` macro is used, with `name` as a string literal. `TracyD3D12ZoneC(ctx, cmdList, name, color)` can be used to create a custom-colored zone. The given command list must be in an open state.

[35]It is considerably faster than the OpenGL's `TracyGpuCollect`.

The macro `TracyD3D12NewFrame(ctx)` is used to mark a new frame, and should appear before or after recording command lists, similar to `FrameMark`. This macro is a key component that enables automatic query data synchronization, so the user doesn't have to worry about synchronizing GPU execution before invoking a collection. Event data can then be collected and sent to the profiler using the `TracyD3D12Collect(ctx)` macro.

3.9.4 OpenCL

OpenCL support is achieved by including the `tracy/TracyOpenCL.hpp` header file. Tracing OpenCL requires the creation of a Tracy OpenCL context using the macro `TracyCLContext(context, device)`, which will return an instance of `TracyCLCtx` object that must be used when creating zones. The specified `device` must be part of the `context`. Cleanup is performed using the `TracyCLDestroy(ctx)` macro. Although not common, it is possible to create multiple OpenCL contexts for the same application.

To mark an OpenCL zone one must make sure that a valid OpenCL `cl_event` object is available. The event will be the object that Tracy will use to query profiling information from the OpenCL driver. For this to work, all OpenCL queues must be created with the `CL_QUEUE_PROFILING_ENABLE` property.

OpenCL zones can be created with the `TracyCLZone(ctx, name)` where `name` will usually be a descriptive name for the operation represented by the `cl_event`. Within the scope of the zone, you must call `TracyCLSetEvent(event)` for the event to be registered in Tracy.

Similarly to Vulkan and OpenGL, you also need to periodically collect the OpenCL events using the `TracyCLCollect(ctx)` macro. A good place to perform this operation is after a `clFinish`, since this will ensure that any previous queued OpenCL commands will have finished by this point.

3.9.5 Multiple zones in one scope

Putting more than one GPU zone macro in a single scope features the same issue as with the `ZoneScoped` macros, described in section 3.4.1 (but this time the variable name is `___tracy_gpu_zone`).

To solve this problem, in case of OpenGL use the `TracyGpuNamedZone` macro in place of `TracyGpuZone` (or the color variant). The same applies to Vulkan and Direct3D 12 – replace `TracyVkZone` with `TracyVkNamedZone` and `TracyD3D12Zone` with `TracyD3D12NamedZone`.

Remember that you need to provide your own name for the created stack variable as the first parameter to the macros.

3.10 Collecting call stacks

Capture of true calls stacks can be performed by using macros with the `S` postfix, which require an additional parameter, specifying the depth of call stack to be captured. The greater the depth, the longer it will take to perform capture. Currently you can use the following macros: `ZoneScopedS`, `ZoneScopedNS`, `ZoneScopedCS`, `ZoneScopedNCS`, `TracyAllocS`, `TracyFreeS`, `TracyMessageS`, `TracyMessageLS`, `TracyMessageCS`, `TracyMessageLCS`, `TracyGpuZoneS`, `TracyGpuZoneCS`, `TracyVkZoneS`, `TracyVkZoneCS`, and the named variants.

Be aware that call stack collection is a relatively slow operation. Table 5 and figure 6 show how long it took to perform a single capture of varying depth on multiple CPU architectures.

You can force call stack capture in the non-`S` postfixed macros by adding the `TRACY_CALLSTACK` define, set to the desired call stack capture depth. This setting doesn't affect the explicit call stack macros.

The maximum call stack depth that can be retrieved is 62 frames. This is a restriction at the level of operating system.

Debugging symbols

To have proper call stack information, the profiled application must be compiled with debugging symbols enabled. You can achieve that in the following way:

Depth	x86	x64	ARM	ARM64
1	34 ns	98 ns	6.62 µs	6.63 µs
2	35 ns	150 ns	8.08 µs	8.25 µs
3	36 ns	168 ns	9.75 µs	10 µs
4	39 ns	190 ns	10.92 µs	11.58 µs
5	42 ns	206 ns	12.5 µs	13.33 µs
10	52 ns	306 ns	19.62 µs	21.71 µs
15	63 ns	415 ns	26.83 µs	30.13 µs
20	77 ns	531 ns	34.25 µs	38.71 µs
25	89 ns	630 ns	41.17 µs	47.17 µs
30	109 ns	735 ns	48.33 µs	55.63 µs
35	123 ns	843 ns	55.87 µs	64.09 µs
40	142 ns	950 ns	63.12 µs	72.59 µs
45	154 ns	1.05 µs	70.54 µs	81 µs
50	167 ns	1.16 µs	78 µs	89.5 µs
55	179 ns	1.26 µs	85.04 µs	98 µs
60	193 ns	1.37 µs	92.75 µs	106.59 µs

Table 5: *Median times of zone capture with call stack. x86, x64: i7 8700K; ARM: Banana Pi; ARM64: ODROID-C2. Selected architectures are plotted on figure 6*

- On MSVC open the project properties and go to *Linker→Debugging→Generate Debug Info*, where the *Generate Debug Information* option should be selected.
- On gcc or clang remember to specify the debugging information `-g` parameter during compilation and omit the strip symbols `-s` parameter. Link the executable with an additional option `-rdynamic` (or `--export-dynamic`, if you are passing parameters directly to the linker).
- On OSX you may need to run `dsymutil` to extract the debugging data out of the executable binary.
- On iOS you will have to add a *New Run Script Phase* to your XCode project, which will execute the following shell script:

```
cp -rf ${TARGET_BUILD_DIR}/${WRAPPER_NAME}.dSYM/* ${TARGET_BUILD_DIR}/${
    UNLOCALIZED_RESOURCES_FOLDER_PATH}/${PRODUCT_NAME}.dSYM
```

  You will also need to setup proper dependencies, by setting the following input file: `${TARGET_BUILD_DIR}/${WRAPPER_NAME}.dSYM`, and the following output file: `${TARGET_BUILD_DIR}/${UNLOCALIZED_RESOURCES_FOLDER_PATH}/${PRODUCT_NAME}.dSYM`.

You may also be interested in symbols from external libraries, especially if you have sampling profiling enabled (section 3.13.4). In MSVC you can retrieve such symbols by going to *Tools→Options→Debugging→Symbols* and selecting appropriate *Symbol file (.pdb) location* servers. Note that additional symbols may significantly increase application startup times.

3.11 Lua support

To profile Lua code using Tracy, include the `tracy/TracyLua.hpp` header file in your Lua wrapper and execute `tracy::LuaRegister(lua_State*)` function to add instrumentation support.

In the Lua code, add `tracy.ZoneBegin()` and `tracy.ZoneEnd()` calls to mark execution zones. You need to call the `ZoneEnd` method, because there is no automatic destruction of variables in Lua and we don't know when the garbage collection will be performed. *Double check if you have included all return paths!*

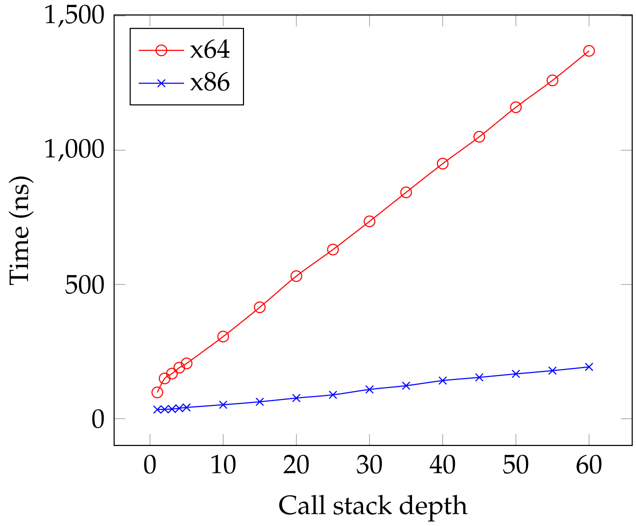

Figure 6: *Plot of call stack capture times (see table 5). Notice that the capture time grows linearly with requested capture depth*

Use `tracy.ZoneBeginN(name)` if you want to set a custom zone name[36].
Use `tracy.ZoneText(text)` to set zone text.
Use `tracy.Message(text)` to send messages.
Use `tracy.ZoneName(text)` to set zone name on a per-call basis.
Lua instrumentation needs to perform additional work (including memory allocation) to store source location. This approximately doubles the data collection cost.

3.11.1 Call stacks

To collect Lua call stacks (see section 3.10), replace `tracy.ZoneBegin()` calls with `tracy.ZoneBeginS(depth)`, and `tracy.ZoneBeginN(name)` calls with `tracy.ZoneBeginNS(name, depth)`. Using the `TRACY_CALLSTACK` macro automatically enables call stack collection in all zones.

Be aware that for Lua call stack retrieval to work, you need to be on a platform which supports collection of native call stacks.

Cost of performing Lua call stack capture is presented in table 6 and figure 7. Lua call stacks include native call stacks, which have a capture cost of their own (table 5) and the `depth` parameter is applied for both captures. The presented data was captured with full Lua stack depth, but only 13 frames were available on the native call stack. Hence, to explain the non-linearity of the graph you need to consider what was really measured:

$$\text{Cost}_{\text{total}}(\text{depth}) = \begin{cases} \text{Cost}_{\text{Lua}}(\text{depth}) + \text{Cost}_{\text{native}}(\text{depth}) & \text{when depth} \leq 13 \\ \text{Cost}_{\text{Lua}}(\text{depth}) + \text{Cost}_{\text{native}}(13) & \text{when depth} > 13 \end{cases}$$

3.11.2 Instrumentation cleanup

Even if Tracy is disabled, you still have to pay the no-op function call cost. To prevent that you may want to use the `tracy::LuaRemove(char* script)` function, which will replace instrumentation calls with white-space. This function does nothing if profiler is enabled.

[36]While technically this name doesn't need to be constant, like in the `ZoneScopedN` macro, it should be, as it is used to group the zones together. This grouping is then used to display various statistics in the profiler. You may still set the per-call name using the `tracy.ZoneName` method.

Depth	Time
1	707 ns
2	699 ns
3	624 ns
4	727 ns
5	836 ns
10	1.77 µs
15	2.44 µs
20	2.51 µs
25	2.98 µs
30	3.6 µs
35	4.33 µs
40	5.17 µs
45	6.01 µs
50	6.99 µs
55	8.11 µs
60	9.17 µs

Table 6: *Median times of Lua zone capture with call stack (x64, 13 native frames)*

3.12 C API

In order to profile code written in C programming language, you will need to include the `tracy/TracyC.h` header file, which exposes the C API.

At the moment there's no support for C API based markup of locks, GPU zones, or Lua.

Important

Tracy is written in C++, so you will need to have a C++ compiler and link with C++ standard library, even if your program is strictly pure C.

Caveats

If you are using MSVC, you will need to disable the *Edit And Continue* feature, for the C API to work[a]. To do so, open the project properties and go to *C/C++*→*General*→*Debug Information Format* and make sure *Program Database for Edit And Continue (/ZI)* is *not* selected.

[a] There's no such requirement for C++ API.

3.12.1 Frame markup

To mark frames, as described in section 3.3, use the following macros:

- `TracyCFrameMark`
- `TracyCFrameMarkNamed(name)`
- `TracyCFrameMarkStart(name)`
- `TracyCFrameMarkEnd(name)`
- `TracyCFrameImage(image, width, height, offset, flip)`

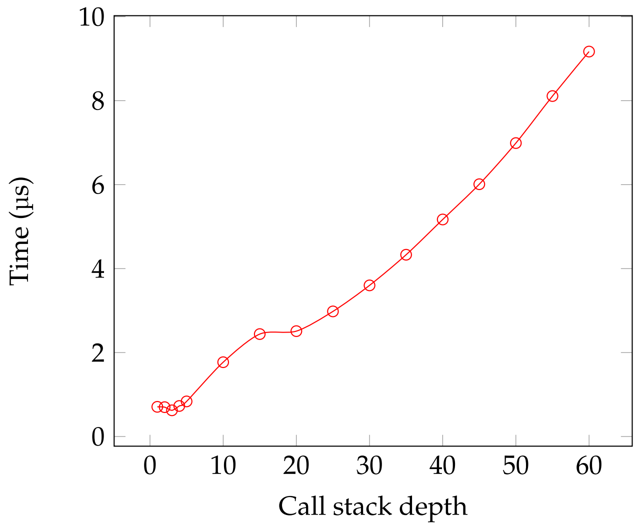

Figure 7: *Plot of call Lua stack capture times (see table 6)*

3.12.2 Zone markup

The following macros mark the beginning of a zone:

- `TracyCZone(ctx, active)`
- `TracyCZoneN(ctx, name, active)`
- `TracyCZoneC(ctx, color, active)`
- `TracyCZoneNC(ctx, name, color, active)`

Refer to sections 3.4 and 3.4.1 for description of macro variants and parameters. The `ctx` parameter specifies the name of a data structure, which will be created on stack to hold the internal zone data.

Unlike C++, there's no automatic destruction mechanism in C, so you will need to manually mark where the zone ends. To do so use the `TracyCZoneEnd(ctx)` macro.

Zone text and name may be set by using the `TracyCZoneText(ctx, txt, size)`, `TracyCZoneValue(ctx, value)` and `TracyCZoneName(ctx, txt, size)` macros. Make sure you are following the zone stack rules, as described in section 3.4.1!

3.12.2.1 Zone context data structure

In typical use cases the zone context data structure is hidden from your view, requiring only to specify its name for the `TracyCZone` and `TracyCZoneEnd` macros. However, it is possible to use it in advanced scenarios, for example if you want to start a zone in one function, then end it in another one. To do so you will need to forward the data structure either through a function parameter, or as a return value, or place it in a thread-local stack structure. To accomplish this you need to keep in mind the following rules:

- The created variable name is exactly what you pass as the `ctx` parameter.
- The data structure is of an opaque, immutable type `TracyCZoneCtx`.
- Contents of the data structure can be copied by assignment. Do not retrieve or use the structure's address – this is asking for trouble.
- You *must* use the data structure (or any of its copies) exactly *once* to end a zone.

3.12.2.2 Zone validation

Since all instrumentation using the C API has to be done by hand, it is possible to miss some code paths where a zone should be started or ended. Tracy will perform additional validation of instrumentation correctness to prevent bad profiling runs. Read section 4.6 for more information.

The validation comes with a performance cost though, which you may not want to pay. If you are *completely sure* that the instrumentation is not broken in any way, you may use the `TRACY_NO_VERIFY` macro, which will disable the validation code.

3.12.2.3 Allocated source locations

Sometimes you might want to provide your own source location data to a zone. For example, you may be integrating Tracy with another programming language, one where there are no guarantees about object lifetime, or how data structures will be laid out in the memory.

To do so, you need to create an *allocated source location*, by calling one of the following functions:

- `___tracy_alloc_srcloc(uint32_t line, const char* source, const char* function)`
- `___tracy_alloc_srcloc_name(uint32_t line, const char* source, const char* function, const char* name, size_t nameSz)`

Where `line` is line number in the `source` source file and `function` is the name of a function in which zone is created. Both `source` and `function` must be null-terminated strings. You may additionally specify an optional zone name, by providing it in the `name` variable, and specifying its size in `nameSz`. None of the provided text strings has to be kept in memory after source location is allocated.

Both functions return an `uint64_t` source location value, which then must be passed to one of the zone begin functions:

- `___tracy_emit_zone_begin_alloc(srcloc, active)`
- `___tracy_emit_zone_begin_alloc_callstack(srcloc, depth, active)`

These functions return a `TracyCZoneCtx` context value, which must be handled, as described in sections 3.12.2 and 3.12.2.1.

The variable representing an allocated source location is of an opaque type. After it is passed to one of the zone begin functions, its value *cannot be reused*. You must allocate a new source location for each zone begin event.

Important

Since you are directly calling the profiler functions here, you will need to take care of manually disabling the code, if the `TRACY_ENABLE` macro is not defined.

3.12.3 Memory profiling

Use the following macros in your implementations of `malloc` and `free`:

- `TracyCAlloc(ptr, size)`
- `TracyCFree(ptr)`

Using this functionality in a proper way can be quite tricky, as you also will need to handle all the memory allocations made by external libraries (which typically allow usage of custom memory allocation functions), but also the allocations made by system functions. If such an allocation can't be tracked, you will need to make sure freeing is not reported[37].

There is no explicit support for `realloc` function. You will need to handle it by marking memory allocations and frees, according to the system manual describing behavior of this routine.

For more information refer to section 3.8.

3.12.4 Plots and messages

To send additional markup in form of plot data points or messages use the following macros:

- `TracyCPlot(name, val)`
- `TracyCMessage(txt, size)`
- `TracyCMessageL(txt)`
- `TracyCMessageC(txt, size, color)`
- `TracyCMessageLC(txt, color)`
- `TracyCAppInfo(txt, size)`

Consult sections 3.6 and 3.7 for more information.

3.12.5 Call stacks

You can collect call stacks of zones and memory allocation events, as described in section 3.10, by using the following `S` postfixed macros: `TracyCZoneS`, `TracyCZoneNS`, `TracyCZoneCS`, `TracyCZoneNCS`, `TracyCAllocS`, `TracyCFreeS`, `TracyCMessageS`, `TracyCMessageLS`, `TracyCMessageCS`, `TracyCMessageLCS`.

3.13 Automated data collection

Tracy will perform automatic collection of system data without user intervention. This behavior is platform specific and may not be available everywhere. Refer to section 2.6 for more information.

3.13.1 CPU usage

System-wide CPU load is gathered with relatively high granularity (one reading every 100 ms). The readings are available as a plot (see section 5.2.3.3). Note that this parameter takes into account all applications running on the system, not only the profiled program.

3.13.2 Context switches

Since the profiled program is executing simultaneously with other applications, you can't have exclusive access to the CPU. The multitasking operating system's scheduler is giving threads waiting to execute short time slices, where part of the work can be done. Afterwards threads are preempted to give other threads a chance to run. This ensures that each program running in the system has a fair environment and no program can hog the system resources for itself.

As a corollary, it is often not enough to know how long it took to execute a zone. The thread in which a zone was running might have been suspended by the system, which artificially increases the time readings.

To solve this problem, Tracy collects context switch[38] information. This data can be then used to see when a zone was in the executing state and where it was waiting to be resumed.

[37] It's not uncommon to see a pattern where a system function returns some allocated memory, which you then need to free.

[38] A context switch happens when any given CPU core stops executing one thread and starts running another one.

Context switch data capture may be disabled by adding the `TRACY_NO_SYSTEM_TRACING` define to the client.

Caveats

- Context switch data is retrieved using the kernel profiling facilities, which are not available to users with normal privilege level. To collect context switches you will need to elevate your rights to admin level, either by running the profiled program from the `root` account on Unix, or through the *Run as administrator* option on Windows. On Android context switches will be collected if you have a rooted device (see section 2.1.6.2 for additional information).
- Android context switch capture requires spawning an elevated process to read kernel data. While the standard `cat` utility can be used for this task, the CPU usage is not acceptable due to how the kernel handles blocking reads. As a workaround, Tracy will inject a specialized kernel data reader program at `/data/tracy_systrace`, which has more acceptable resource requirements.

3.13.3 CPU topology

Tracy may perform discovery of CPU topology data in order to provide further information about program performance characteristics. It is very useful when combined with context switches (section 3.13.2).

In essence, the topology information gives you context about what any given *logical CPU* really is and how it relates to other logical CPUs. The topology hierarchy consists of packages, cores and threads.

Packages contain cores and shared resources, such as memory controller, L3 cache, etc. A store-bought CPU is an example of a package. While you may think that multi-package configurations would be a domain of servers, they are actually quite common in the mobile devices world, with many platforms using the *big.LITTLE* arrangement of two packages.

Cores contain at least one thread and shared resources: execution units, L1 and L2 cache, etc.

Threads (or *logical CPUs*; not to be confused with program threads) are basically the processor instruction pipelines. A pipeline might become stalled, for example due to pending memory access, leaving core resources unused. To reduce this bottleneck, some CPUs may use simultaneous multithreading[39], in which more than one pipeline will be using a single physical core resources.

Knowing which package and core any logical CPU belongs to enables many insights. For example, two threads scheduled to run on the same core will compete for shared execution units and cache, resulting in reduced performance. Or, a migration of a program thread from one core to another core will invalidate L1 and L2 cache, which is less costly than a migration from one package to another, which also invalidates L3 cache.

Important

In this manual, the word *core* is typically used as a short term for *logical CPU*. Do not confuse it with physical processor cores.

3.13.4 Call stack sampling

Manual markup of zones doesn't cover every function existing in a program and cannot be performed in system libraries, or in kernel. This can leave blank spaces on the trace, leaving you with no clue what the application was doing. Tracy is able to periodically inspect state of running threads, providing you with a snapshot of call stack at the time when sampling was performed. While this information doesn't have the fidelity of manually inserted zones, it can sometimes give you an insight where to go next.

This feature requires privilege elevation, as described in chapter 3.13.2. Proper setup of the required program debugging data is described in chapter 3.10.

[39] Commonly known as Hyper-threading.

3.13.5 Executable code retrieval

To enable deep insight into program execution, Tracy will capture small chunks of the executable image during profiling. The retrieved code can be subsequently disassembled to be inspected in detail. This functionality will be performed only for functions that are no larger than 64 KB and only if symbol information is present.

You should be extra careful when working with non-public code, as parts of your program will be embedded in the captured trace.

3.14 Trace parameters

Sometimes it is desired to change how the profiled application is behaving during the profiling run, for example you may want to enable or disable capture of frame images without recompiling and restarting your program. To be able to do so you must register a callback function using the `TracyParameterRegister(callback)` macro, where `callback` is a function conforming to the following signature:

```
void Callback(uint32_t idx, int32_t val)
```

The `idx` argument is an user-defined parameter index and `val` is the value set in the profiler user interface.

To specify individual parameters, use the `TracyParameterSetup(idx, name, isBool, val)` macro. The `idx` value will be passed to the callback function for identification purposes (Tracy doesn't care what it's set to). `Name` is the parameter label, displayed on the list of parameters. Finally, `isBool` determines if `val` should be interpreted as a boolean value, or as an integer number.

Important

Usage of trace parameters makes profiling runs dependent on user interaction with the profiler, and thus it's not recommended to be employed if a consistent profiling environment is desired. Furthermore, interaction with the parameters is only possible in the graphical profiling application, and not in the command line capture utility.

4 Capturing the data

After the client application has been instrumented, you will want to connect to it using a server, which is available either as a headless capture-only utility, or as a full-fledged graphical profiling interface.

4.1 Command line

You can capture a trace using a command line utility contained in the `capture` directory. To use it you will need to provide the following parameters:

- `-o output.tracy` – the file name of the resulting trace.
- `-a address` – specifies the IP address (or a domain name) of the client application (uses `localhost` if not provided).
- `-p port` – network port which should be used (optional).

If there is no client running at the given address, the server will wait until a connection can be made. During the capture the following information will be displayed:

```
% ./capture -a 127.0.0.1 -o trace
Connecting to 127.0.0.1:8086...
Queue delay: 5 ns
Timer resolution: 3 ns
   1.33 Mbps / 40.4% = 3.29 Mbps | Net: 64.42 MB | Mem: 283.03 MB | Time: 10.6 s
```

The *queue delay* and *timer resolution* parameters are calibration results of timers used by the client. The next line is a status bar, which displays: network connection speed, connection compression ratio, and the resulting uncompressed data rate; total amount of data transferred over the network; memory usage of the capture utility; time extent of the captured data.

You can disconnect from the client and save the captured trace by pressing Ctrl + C.

4.2 Interactive profiling

If you want to look at the profile data in real-time (or load a saved trace file), you can use the data analysis utility contained in the `profiler` directory. After starting the application, you will be greeted with a welcome dialog (figure 8), presenting a bunch of useful links (*User manual*, *Web*, *Join chat* and *Sponsor*). The *Web* button opens a drop-down list with links to the profiler's *Home page* and a bunch of *Feature videos*.

The client *address entry* field and the *Connect* button are used to connect to a running client[40]. You can use the connection history button to display a list of commonly used targets, from which you can quickly select an address. You can remove entries from this list by hovering the mouse cursor over an entry and pressing the Del. button on the keyboard.

If you want to open a trace that you have stored on the disk, you can do so by pressing the *Open saved trace* button.

The *discovered clients* list is only displayed if there are clients broadcasting their presence on the local network[41]. Each entry shows the address[42] of the client (and optionally port, if different from the default one), how long the client has been running, and the name of the application that is profiled. Clicking on an entry will connect to the client. Incompatible clients are grayed-out and can't be connected to. Clicking on the *Filter* toggle button will display client filtering input fields, allowing removal of the displayed entries, according to their address, port number, or program name. If filters are active, a yellow warning icon will be displayed.

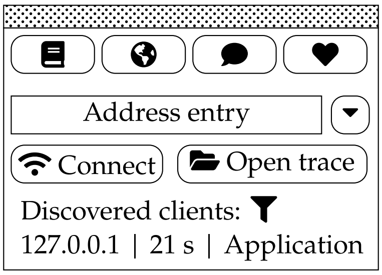

Figure 8: *Welcome dialog.*

Both connecting to a client and opening a saved trace will present you with the main profiler view, which you can use to analyze the data (see section 5).

4.2.1 Connection information pop-up

If this is a real-time capture, you will also have access to the connection information pop-up (figure 9) through the *Connection* button, with the capture status similar to the one displayed by the command line utility.

[40]Note that a custom port may be provided here, for example by entering '127.0.0.1:1234'.
[41]Only on IPv4 networks and only within the broadcast domain.
[42]Either as an IP address, or as a host name, if able to resolve.

This dialog also displays the connection speed graphed over time and the profiled application's current frames per second and frame time measurements. The *Query backlog* consists of two numbers. The first one represents the number of queries that were held back due to the bandwidth volume overwhelming the available network send buffer. The second one shows how many queries are in-flight, meaning requests which were sent to the client, but weren't yet answered. While these numbers drains down to zero, the performance of real time profiling may be temporarily compromised. The circle displayed next to the bandwidth graph signals the connection status. If it's red, the connection is active. If it's gray, the client has disconnected.

You can use the *Save trace* button to save the current profile data to a file[43]. Use the *Stop* button to disconnect from the client[44]. The *Discard* button is used to discard current trace.

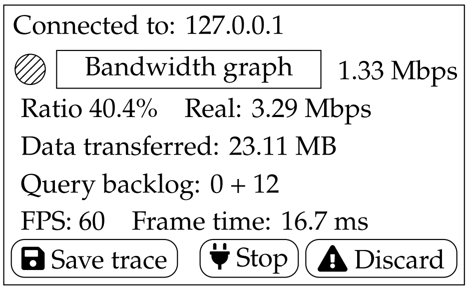

Figure 9: *Connection information pop-up.*

If the profiled application opted to provide trace parameters (see section 3.14) and the connection is still active, this pop-up will also contain a *trace parameters* section, listing all the provided options. When you change any value here, a callback function will be executed on the client.

4.2.2 Automatic loading or connecting

You can pass trace file name as an argument to the profiler application to open the capture, skipping the welcome dialog. You can also use the `-a address` argument to automatically connect to the given address. To specify the network port, pass the `-p port` parameter. It will be used for connections to client (overridable in the UI) and for listening to client discovery broadcasts.

4.3 Connection speed

Tracy network bandwidth requirements depend on the amount of data collection the profiled application is performing. In typical use case scenarios, you may expect anything between 1 Mbps and 100 Mbps data transfer rate.

The maximum attainable connection speed is determined by the ability of the client to provide data and the ability of the server to process the received data. In an extreme conditions test performed on an i7 8700K, the maximum transfer rate peaked at 950 Mbps. In each second the profiler was able to process 27 million zones and consume 1 GB of RAM.

4.4 Memory usage

The captured data is stored in RAM and only written to the disk, when the capture finishes. This can result in memory exhaustion when you are capturing massive amounts of profile data, or even in normal usage situations, when the capture is performed over a long stretch of time. The recommended usage pattern is to perform moderate instrumentation of the client code and limit capture time to the strict necessity.

[43]This should be taken literally. If a live capture is in progress and a save is performed, some data may be missing from the capture and won't be saved.

[44]While requesting disconnect stops retrieval of any new events, the profiler will wait for any data that is still pending for the current set of events.

In some cases it may be useful to perform an *on-demand* capture, as described in section 2.1.2. In such case you will be able to profile only the interesting case (e.g. behavior during loading of a level in a game), ignoring all the unneeded data.

If you truly need to capture large traces, you have two options. Either buy more RAM, or use a large swap file on a fast disk drive[45].

4.5 Trace versioning

Each new release of Tracy changes the internal format of trace files. While there is a backwards compatibility layer, allowing loading of traces created by previous versions of Tracy in new releases, it won't be there forever. You are thus advised to upgrade your traces using the utility contained in the `update` directory.

To use it, you will need to provide the input file and the output file. The program will print a short summary when it finishes, with information about trace file versions, their respective sizes and the output trace file compression ratio:

```
% ./update old.tracy new.tracy
old.tracy (0.3.0) {916.4 MB} -> new.tracy (0.4.0) {349.4 MB, 31.53%}  9.7 s, 38.13% change
```

The new file contains the same data as the old one, but in the updated internal representation. Note that to perform an upgrade, whole trace needs to be loaded to memory.

4.5.1 Archival mode

The update utility supports optional higher levels of data compression, which reduce disk size of traces, at the cost of increased compression times. With the default settings, the output files have a reasonable size and are quick to save and load. A list of available compression modes, parameters that enable them, and compression results is available in table 7 and figures 10, 11 and 12.

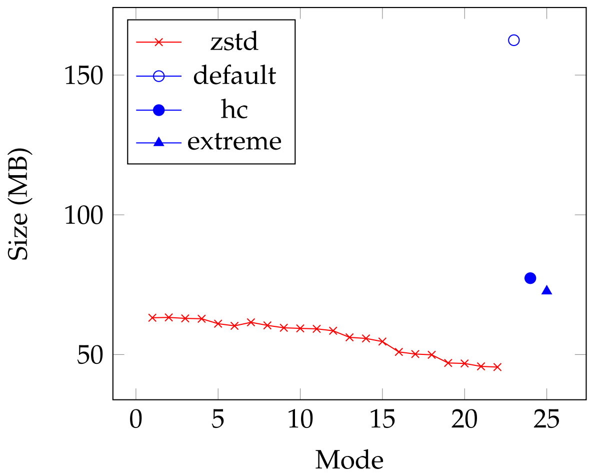

Figure 10: *Plot of trace sizes for different compression modes (see table 7).*

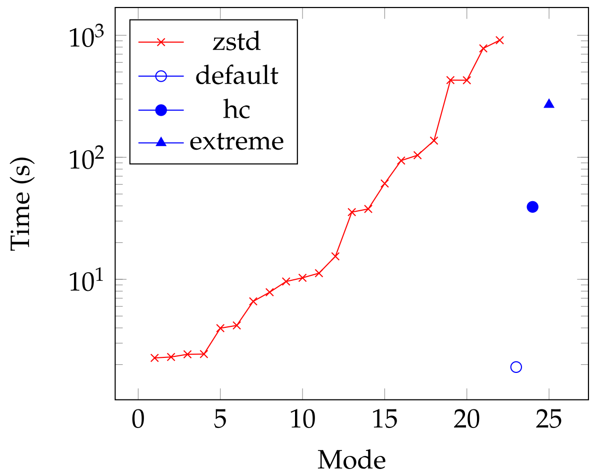

Figure 11: *Logarithmic plot of trace compression times for different compression modes (see table 7).*

[45]The operating system is able to manage memory paging much better than Tracy would be ever able to.

Mode	Size	Ratio	Save time	Load time
default	162.48 MB	17.19%	1.91 s	470 ms
`--hc`	77.33 MB	8.18%	39.24 s	401 ms
`--extreme`	72.67 MB	7.68%	4:30	406 ms
`--zstd 1`	63.17 MB	6.68%	2.27 s	868 ms
`--zstd 2`	63.29 MB	6.69%	2.31 s	884 ms
`--zstd 3`	62.94 MB	6.65%	2.43 s	867 ms
`--zstd 4`	62.81 MB	6.64%	2.44 s	855 ms
`--zstd 5`	61.04 MB	6.45%	3.98 s	855 ms
`--zstd 6`	60.27 MB	6.37%	4.19 s	827 ms
`--zstd 7`	61.53 MB	6.5%	6.6 s	761 ms
`--zstd 8`	60.44 MB	6.39%	7.84 s	746 ms
`--zstd 9`	59.58 MB	6.3%	9.6 s	724 ms
`--zstd 10`	59.36 MB	6.28%	10.29 s	706 ms
`--zstd 11`	59.2 MB	6.26%	11.23 s	717 ms
`--zstd 12`	58.51 MB	6.19%	15.43 s	695 ms
`--zstd 13`	56.16 MB	5.94%	35.55 s	642 ms
`--zstd 14`	55.76 MB	5.89%	37.74 s	627 ms
`--zstd 15`	54.65 MB	5.78%	1:01	600 ms
`--zstd 16`	50.94 MB	5.38%	1:34	537 ms
`--zstd 17`	50.18 MB	5.30%	1:44	542 ms
`--zstd 18`	49.91 MB	5.28%	2:17	554 ms
`--zstd 19`	46.99 MB	4.97%	7:09	605 ms
`--zstd 20`	46.81 MB	4.95%	7:08	608 ms
`--zstd 21`	45.77 MB	4.84%	13:01	614 ms
`--zstd 22`	45.52 MB	4.81%	15:11	621 ms

Table 7: *Compression results for an example trace. Tests performed on Ryzen 9 3900X.*

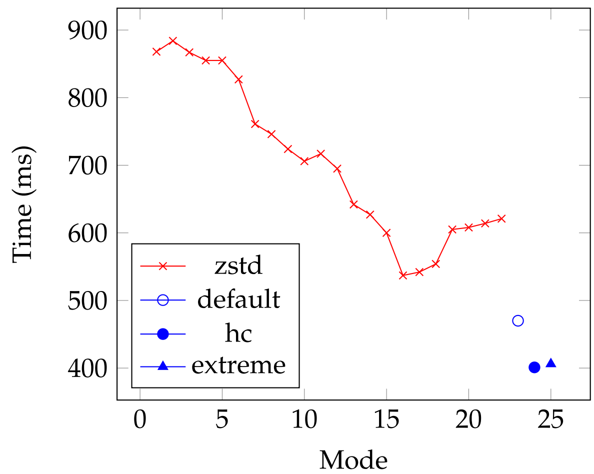

Figure 12: *Plot of trace load times for different compression modes (see table 7).*

Trace files created using the *default*, *hc* and *extreme* modes are optimized for fast decompression and can be further compressed using file compression utilities. For example, using 7-zip results in archives of the following sizes: 77.2 MB, 54.3 MB, 52.4 MB.

For archival purposes it is however much better to use the *zstd* compression modes, which are faster, compress trace files more tightly, and are directly loadable by the profiler, without the intermediate decompression step.

4.6 Instrumentation failures

In some cases your program may be incorrectly instrumented, for example you could have unbalanced zone begin and end events, or you could report a memory free event without first reporting a memory allocation event. When Tracy detects such misbehavior it immediately terminates connection with the client and displays an error message.

5 Analyzing captured data

You have instrumented your application and you have captured a profiling trace. Now you want to look at the collected data. You can do this in the application contained in the `profiler` directory.

The workflow is identical, whether you are viewing a previously saved trace, or if you're performing a live capture, as described in section 4.2.

5.1 Time display

In most cases Tracy will display an approximation of time value, depending on how big it is. For example, a short time range will be displayed as 123 ns, and some longer ones will be shortened to 123.45 µs, 123.45 ms, 12.34 s, 1:23.4, 12:34:56, or even 1d12:34:56 to indicate more than a day has passed.

While such presentation makes time values easy to read, it is not always appropriate. For example, you may have multiple events happen at a time approximated to 1:23.4, giving you a precision of only 1/10 of a second. There's certainly a lot that can happen in 100 ms.

To solve this problem, an alternative time display is used in appropriate places. It combines a day–hour–minute–second value with full nanosecond resolution, resulting in values such as 1:23 456,789,012 ns.

5.2 Main profiler window

The main profiler window is split into three sections, as seen on figure 13: the control menu, the frame time graph and the timeline display.

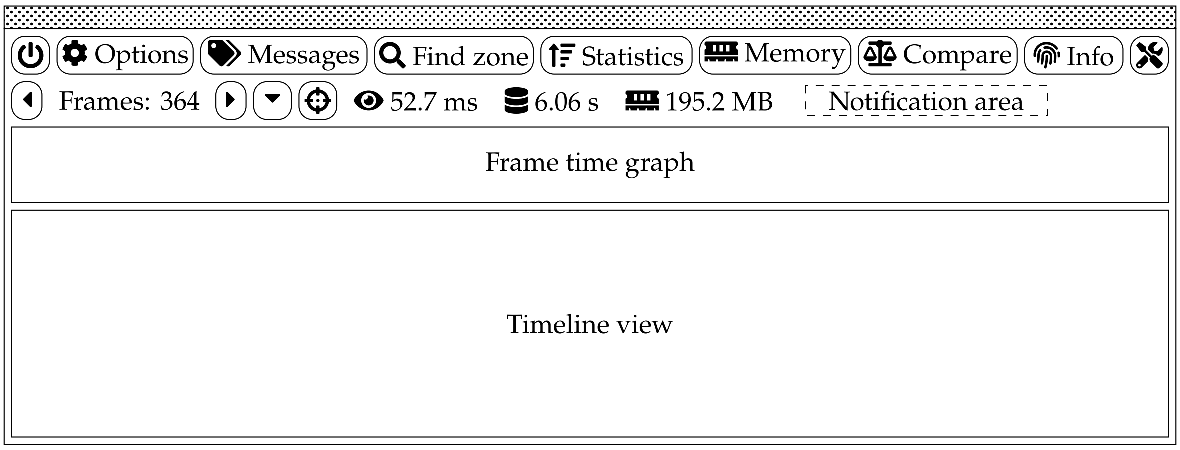

Figure 13: *Main profiler window. Note that the top line of buttons has been split into two rows in this manual.*

5.2.1 Control menu

The control menu (top row of buttons) provides access to various features of the profiler. The buttons perform the following actions:

- *Connection* – Opens the connection information popup (see section 4.2.1). Only available when live capture is in progress.
- *Close* – This button unloads the current profiling trace and returns to the welcome menu, where another trace can be loaded. In live captures it is replaced by *Pause*, *Resume* and *Stopped* buttons.
- *Pause* – While a live capture is in progress, the profiler will display the last three fully captured frames, so that you can see the current behavior of the program. Use this button[46] to stop the automatic updates of the timeline view (the capture will be still progressing).
- *Resume* – Use this button to resume following the most recent three frames in a live capture.
- *Stopped* – Inactive button used to indicate that the client application was terminated.
- *Options* – Toggles the settings menu (section 5.3).
- *Messages* – Toggles the message log window (section 5.4), which displays custom messages sent by the client, as described in section 3.7.
- *Find zone* – This buttons toggles the find zone window, which allows inspection of zone behavior statistics (section 5.6).
- *Statistics* – Toggles the statistics window, which displays zones sorted by their total time cost (section 5.5).
- *Memory* – Various memory profiling options may be accessed here (section 5.8).
- *Compare* – Toggles the trace compare window, which allows you to see the performance difference between two profiling runs (section 5.7).
- *Info* – Show general information about the trace (section 5.11).
- *Tools* – Allows access to optional data collected during capture. Some choices might be unavailable.
  - *Playback* – If frame images were captured (section 3.3.3), you will have option to open frame image playback window, described in chapter 5.17.
  - *CPU data* – If context switch data was captured (section 3.13.2), this button will allow inspecting what was the processor load during the capture, as described in section 5.18.
  - *Annotations* – If annotations have been made (section 5.2.5), you can open a list of all annotations, described in chapter 5.20.

The frame information block consists of four elements: the current frame set name along with the number of captured frames, the two navigational buttons and , which allow you to focus the timeline view on the previous or next frame, and the frame set selection button , which is used to switch to a another frame set[47]. The *Go to frame* button allows zooming the timeline view on the specified frame. For more information about marking frames, see section 3.3.

The next three items show the *view time range*, the *time span* of the whole capture, and the *memory usage* of the profiler.

[46]Or perform any action on the timeline view.
[47]See section 5.2.3.2 for another way to change the active frame set.

5.2.1.1 Notification area

The notification area is used to display informational notices, for example how long it took to load a trace from disk. A pulsating dot next to the icon indicates that some background tasks are being performed, that may need to be completed before full capabilities of the profiler are available. If a crash was captured during profiling (section 2.5), a *crash* icon will be displayed. The red icon indicates that queries are currently being backlogged, while the same yellow icon indicates that some queries are currently in-flight (see chapter 4.2.1 for more information).

If drawing of timeline elements was disabled in the options menu (section 5.3), the following orange icons will be used to remind the user about that fact. Click on the icons to enable drawing of the selected elements. Note that collapsed labels (section 5.2.3.3) are not taken into account here.

- – Display of empty labels is enabled.
- – Context switches are hidden.
- – CPU data is hidden.
- – GPU zones are hidden.
- – CPU zones are hidden.
- – Locks are hidden.
- – Plots are hidden.
- – Ghost zones are not displayed.
- – At least one timeline item (e.g. a single thread, a single plot, a single lock, etc.) is hidden.

5.2.2 Frame time graph

The graph of currently selected frame set (figure 14) provides an outlook on the time spent in each frame, allowing you to see where the problematic frames are and to quickly navigate to them.

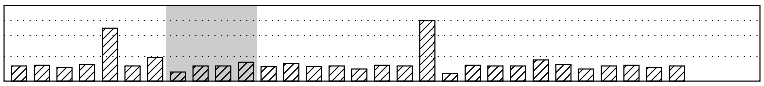

Figure 14: *Frame time graph.*

Each bar displayed on the graph represents an unique frame in the current frame set[48]. The progress of time is in the right direction. The height of the bar indicates the time spent in frame, complemented with the color information:

- If the bar is *blue*, then the frame met the *best* time of 143 FPS, or 6.99 ms[49] (represented by blue target line).
- If the bar is *green*, then the frame met the *good* time of 59 FPS, or 16.94 ms (represented by green target line).
- If the bar is *yellow*, then the frame met the *bad* time of 29 FPS, or 34.48 ms (represented by yellow target line).
- If the bar is *red*, then the frame didn't met any time limits.

[48]Unless the view is zoomed out and multiple frames are merged into one column.
[49]The actual target is 144 FPS, but one frame leeway is allowed to account for timing inaccuracies.

The frames visible on the timeline are marked with a violet box drawn over them.

When a zone is displayed in the find zone window (section 5.6), the coloring of frames may be changed, as described in section 5.6.2.

Moving the mouse cursor over the frames displayed on the graph will display tooltip with information about frame number, frame time, frame image (if available, see chapter 3.3.3), etc. Such tooltips are common for many UI elements in the profiler and won't be mentioned later in the manual.

The timeline view may be focused on the frames, by clicking or dragging the left mouse button on the graph. The graph may be scrolled left and right by dragging the right mouse button over the graph. The view may be zoomed in and out by using the mouse scroll. If the view is zoomed out, so that multiple frames are merged into one column, the highest frame time will be used to represent the given column.

Clicking the left mouse button on the graph while the Ctrl key is pressed will open the frame image playback window (section 5.17) and set the playback to the selected frame. See section 3.3.3 for more information about frame images.

5.2.3 Timeline view

The timeline is the most important element of the profiler UI. All the captured data is displayed there, laid out on the horizontal axis, according to the flow of time. Where there was no profiling performed, the timeline is dimmed out. The view is split into three parts: the time scale, the frame sets and the combined zones, locks and plots display.

Collapsed items Due to extreme differences in time scales, you will almost constantly see events that are too small to be displayed on the screen. Such events have preset minimum size (so they can be seen) and are marked with a zig-zag pattern, to indicate that you need to zoom-in to see more detail.

The zig-zag pattern can be seen applied to frame sets on figure 16, and to zones on figure 17.

5.2.3.1 Time scale

The time scale is a quick aid in determining the relation between screen space and the time it represents (figure 15).

Figure 15: *Time scale.*

The leftmost value on the scale represents the time at which the timeline starts. The rest of numbers label the notches on the scale, with some numbers omitted, if there's no space to display them.

5.2.3.2 Frame sets

Frames from each frame set are displayed directly underneath the time scale. Each frame set occupies a separate row. The currently selected frame set is highlighted with bright colors, with the rest dimmed out.

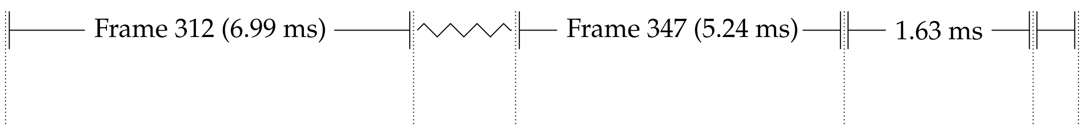

Figure 16: *Frames on the timeline.*

On figure 16 we can see the fully described frames 312 and 347. The description consists of the frame name, which is *Frame* for the default frame set (section 3.3) or the name you used for the secondary name set (section 3.3.1), the frame number and the frame time. The frame 348 is too small to be fully displayed, so only the frame time is shown. The frame 349 is even smaller, with no space for any text. Moreover, frames 313 to 346 are too small to be displayed individually, so they are replaced with a zig-zag pattern, as described in section 5.2.3.

You can also see that there are frame separators, projected down to the rest of the timeline view. Note that only the separators for the currently selected frame set are displayed. You can make a frame set active by clicking the left mouse button on a frame set row you want to select (also see section 5.2.1).

Clicking the middle mouse button on a frame will zoom the view to the extent of the frame.

If a frame has an associated frame image (see chapter 3.3.3), you can hold the Ctrl key and click the left mouse button on the frame, to open the frame image playback window (see chapter 5.17) and set the playback to the selected frame.

5.2.3.3 Zones, locks and plots display

On this combined view you will find the zones with locks and their associated threads. The plots are graphed right below.

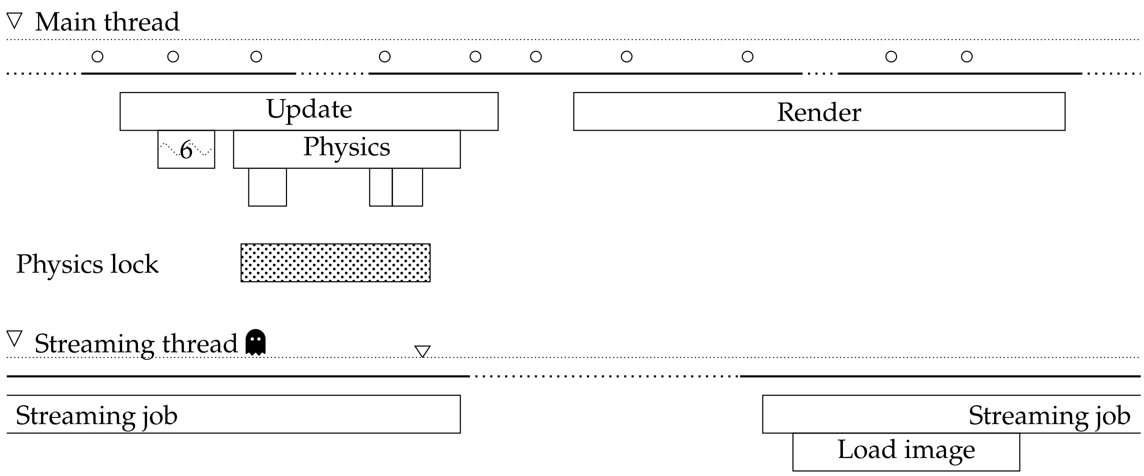

Figure 17: *Zones and locks display.*

The left hand side *index area* of the timeline view displays various labels (threads, locks), which can be categorized in the following way:

- *Light blue label* – GPU context. Multi-threaded Vulkan, OpenCL and Direct3D 12 contexts are additionally split into separate threads.
- *Pink label* – CPU data graph.
- *White label* – A CPU thread. Will be replaced by a bright red label in a thread that has crashed (section 2.5). If automated sampling was performed, clicking the left mouse button on the *ghost zones* button will switch zone display mode between 'instrumented' and 'ghost'.
- *Light red label* – Indicates a lock.
- *Yellow label* – Plot.

Labels accompanied by the symbol can be collapsed out of the view, to reduce visual clutter. Hover the mouse pointer over the label to display additional information. Click the middle mouse button on a label to zoom the view to the extent of the label contents.

Zones In an example on figure 17 you can see that there are two threads: *Main thread* and *Streaming thread*[50]. We can see that the *Main thread* has two root level zones visible: *Update* and *Render*. The *Update* zone is split into further sub-zones, some of which are too small to be displayed at the current zoom level. This is indicated by drawing a zig-zag pattern over the merged zones box (section 5.2.3), with the number of collapsed zones printed in place of zone name. We can also see that the *Physics* zone acquires the *Physics lock* mutex for the most of its run time.

Meanwhile the *Streaming thread* is performing some *Streaming jobs*. The first *Streaming job* sent a message (section 3.7), which in addition to being listed in the message log is being indicated by a triangle over the thread separator. When there are multiple messages in one place, the triangle outline shape changes to a filled triangle.

At high zoom levels, the zones will be displayed with additional markers, as presented on figure 18. The red regions at the start and end of a zone indicate the cost associated with recording an event (*Queue delay*). The error bars show the timer inaccuracy (*Timer resolution*). Note that these markers are only *approximations*, as there are many factors that can impact the true cost of capturing a zone, for example cache effects, or CPU frequency scaling, which is unaccounted for (see section 2.2.2).

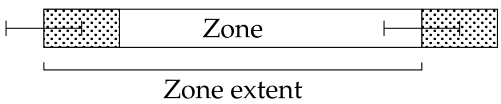

Figure 18: *Approximation of timer inaccuracies and zone collection cost.*

The GPU zones are displayed just like CPU zones, with an OpenGL/Vulkan/Direct3D/OpenCL context in place of a thread name.

Hovering the mouse pointer over a zone will highlight all other zones that have the same source location with a white outline. Clicking the left mouse button on a zone will open zone information window (section 5.12). Holding the Ctrl key and clicking the left mouse button on a zone will open zone statistics window (section 5.6). Clicking the middle mouse button on a zone will zoom the view to the extent of the zone.

Ghost zones Display of ghost zones (not pictured on figure 17, but similar to normal zones view) can be enabled by clicking on the *ghost zones* icon next to thread label, available if automated sampling (see chapter 3.13.4) was performed. Ghost zones will also be displayed by default, if no instrumented zones are available for a given thread, to help with pinpointing functions that should be instrumented.

Ghost zones represent true function calls in the program, periodically reported by the operating system. Due to the limited resolution of sampling, you need to take great care when looking at reported timing data. While it may be apparent that some small function requires a relatively long time to execute, for example 125 µs (8 kHz sampling rate), in reality this time represents a period between taking two distinct samples, not the actual function run time. Similarly, two (or more) distinct function calls may be represented as a single ghost zone, because the profiler doesn't have the information needed to know about true lifetime of a sampled function.

Another common pitfall to watch for is the order of presented functions. *It is not what you expect it to be!* Read chapter 5.13.1 for a critical insight on how call stacks might seem nonsensical at first, and why they aren't.

The available information about ghost zones is quite limited, but it's enough to give you a rough outlook on the execution of your application. The timeline view alone is more than any other statistical profiler is able to present. In addition to that, Tracy properly handles inlined function calls, which are indicated by darker colored ghost zones.

Clicking the left mouse button on a ghost zone will open the corresponding source file location, if able (see chapter 5.15 for conditions). There are three ways in which source locations can be assigned to a ghost

[50]By clicking on a thread name you can temporarily disable display of the zones in this thread.

zone:

1. If the selected ghost zone is *not* an inline frame and its symbol data has been retrieved, the source location points to the function entry location (first line of the function).
2. If the selected ghost zone is *not* an inline frame, but its symbol data is not available, the source location will point to a semi-random location within the function body (i.e. to one of the sampled addresses in the program, but not necessarily the one representing the selected time stamp, as multiple samples with different addresses may be merged into one ghost zone).
3. If the selected ghost zone *is* an inline frame, the source location will point to a semi-random location within the inlined function body (see details in the above point). It is not possible to go to the entry location of such function, as it doesn't exist in the program binary. Inlined functions begin in the parent function.

Call stack samples The row of dots right below the *Main thread* label shows call stack sample points, which may have been automatically captured (see chapter 3.13.4 for more detail). Hovering the mouse pointer over each dot will display a short call stack summary, while clicking on a dot with the left mouse button will open a more detailed call stack information window (see section 5.13).

Context switches The thick line right below the samples represents context switch data (see section 3.13.2). We can see that the main thread, as displayed, starts in a suspended state, represented by the dotted region. Then it is woken up and starts execution of the `Update` zone. In midst of the physics processing it is preempted, which explains why there is an empty space between child zones. Then it is resumed again and continues execution into the `Render` zone, where it is preempted again, but for a shorter time. After rendering is done, the thread sleeps again, presumably waiting for the vertical blanking, to indicate next frame. Similar information is also available for the streaming thread.

Context switch regions are using the following color key:

- *Green* – Thread is running.
- *Red* – Thread is waiting to be resumed by the scheduler. There are many reasons why a thread may be in the waiting state. Hovering the mouse pointer over the region will display more information.
- *Blue* – Thread is waiting to be resumed and is migrating to another CPU core. This might have visible performance effects, because low level CPU caches are not shared between cores, which may result in additional cache misses. To avoid this problem, you may pin a thread to a specific core, by setting its affinity.
- *Bronze* – Thread has been placed in the scheduler's run queue and is about to be resumed.

CPU data This label is only available if context switch data was collected. It is split into two parts: a graph of CPU load by various threads running in the system, and a per-core thread execution display.

The CPU load graph is showing how much CPU resources were used at any given time during program execution. The green part of the graph represents threads belonging to the profiled application and the gray part of the graph shows all other programs running in the system. Hovering the mouse pointer over the graph will display a list of threads running on the CPU at the given time.

Each line in the thread execution display represents a separate logical CPU thread. If CPU topology data is available (see section 3.13.3), package and core assignment will be displayed in brackets, in addition to numerical processor identifier (i.e. `[`*`package`*`:`*`core`*`] CPU `*`thread`*). When a core is busy executing a thread, a zone will be drawn at the appropriate time. Zones are colored according to the following key:

- *Bright color* – or *orange* if dynamic thread colors are disabled – Thread tracked by the profiler.

- *Dark blue* – Thread existing in the profiled application, but not known to the profiler. This may include internal profiler threads, helper threads created by external libraries, etc.
- *Gray* – Threads assigned to other programs running in the system.

When the mouse pointer is hovered over either the CPU data zone, or the thread timeline label, Tracy will display a line connecting all zones associated with the selected thread. This can be used to easily see how the thread was migrating across the CPU cores.

Careful examination of the data presented on this graph may allow you to determine areas where the profiled application was fighting for system resources with other programs (see section 2.2.1), or give you a hint to add more instrumentation macros.

Locks Mutual exclusion zones are displayed in each thread that tries to acquire them. There are three color-coded kinds of lock event regions that may be displayed. Note that when the timeline view is zoomed out, the contention regions are always displayed over the uncontented ones.

- *Green region*[51] – The lock is being held solely by one thread and no other thread tries to access it. In case of shared locks it is possible that multiple threads hold the read lock, but no thread requires a write lock.
- *Yellow region* – The lock is being owned by this thread and some other thread also wants to acquire the lock.
- *Red region* – The thread wants to acquire the lock, but is blocked by other thread, or threads in case of shared lock.

Hovering the mouse pointer over a lock timeline will highlight the lock in all threads to help reading the lock behavior. Hovering the mouse pointer over a lock event will display important information, for example a list of threads that are currently blocking, or which are blocked by the lock. Clicking the left mouse button on a lock event or a lock label will open the lock information window, as described in section 5.16. Clicking the middle mouse button on a lock event will zoom the view to the extent of the event.

Plots The numerical data values (figure 19) are plotted right below the zones and locks. Note that the minimum and maximum values currently displayed on the plot are visible on the screen, along with the y range of the plot and number of drawn data points. The discrete data points are indicated with little rectangles. Multiple data points are indicated by a filled rectangle.

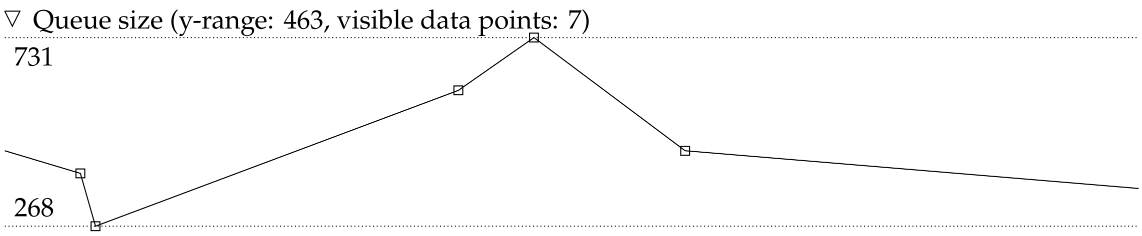

Figure 19: *Plot display.*

When memory profiling (section 3.8) is enabled, Tracy will automatically generate a *Memory usage* plot, which has extended capabilities. Hovering over a data point (memory allocation event) will visually display duration of the allocation. Clicking the left mouse button on the data point will open the memory

[51]This region type is disabled by default and needs to be enabled in options (section 5.3).

allocation information window, which will display the duration of the allocation as long as the window is open.

Another plot that is automatically provided by Tracy is the *CPU usage* plot, which represents the total system CPU usage percentage (it is not limited to the profiled application).

5.2.4 Navigating the view

Hovering the mouse pointer over the timeline view will display a vertical line that can be used to visually line-up events in multiple threads. Dragging the left mouse button will display time measurement of the selected region.

The timeline view may be scrolled both vertically and horizontally by dragging the right mouse button. Note that only the zones, locks and plots scroll vertically, while the time scale and frame sets always stay in place.

You can zoom in and out the timeline view by using the mouse scroll. You can select a range to which you want to zoom-in by dragging the middle mouse button. Dragging the middle mouse button while the Ctrl key is pressed will zoom-out.

5.2.5 Annotating the trace

Sometimes you may want to add notes to a trace. For example, you may want to mark a region to ignore, because the application was out-of-focus, or a region where a new user was connecting to the game, which resulted in a frame drop that needs to be investigated.

To add an annotation, drag the left mouse button over the timeline view, while holding the Ctrl key. When the mouse key is released, a new annotation region will be added and a settings window will open (section 5.19), allowing you to enter a description.

Annotations are displayed on the timeline, as presented on figure 20. Clicking on the circle next to the text description will open the annotation settings window, in which you can modify or remove the region.

Figure 20: *Annotation region.*

Please note that while the annotations persist between profiling sessions, they are not saved in the trace, but in the user data files, as described in section 7.2.

5.3 Options menu

In this window you can set various trace-related options. The timeline view might sometimes become overcrowded, in which case disabling display of some profiling events can increase readability.

- *Draw empty labels* – By default threads that don't have anything to display at the current zoom level are hidden. Enabling this option will show them anyway.
- *Draw context switches* – Allows disabling context switch display in threads.
  - *Darken inactive thread* – If enabled, inactive regions in threads will be dimmed out.
- *Draw CPU data* – Per-CPU behavior graph can be disabled here.
  - *Draw CPU usage graph* – You can disable drawing of the CPU usage graph here.

- *Draw GPU zones* – Allows disabling display of OpenGL/Vulkan/Direct3D/OpenCL zones. The *GPU zones* drop-down allows disabling individual GPU contexts and setting CPU/GPU drift offsets (see section 3.9 for more information). The *Auto* button automatically measures the GPU drift value[52].
- *Draw CPU zones* – Determines whether CPU zones are displayed.
  - *Draw ghost zones* – Controls if ghost zones should be displayed in threads which don't have any instrumented zones available.
  - *Zone colors* – Zones with no user-set color may be colored according to the following schemes:
    - *Disabled* – A constant color (blue) will be used.
    - *Thread dynamic* – Zones are colored according to a thread (identifier number) they belong to and depth level.
    - *Source location dynamic* – Zone color is determined by source location (function name) and depth level.
  - *Namespaces* – controls display behavior of long zone names, which don't fit inside a zone box:
    - *Full* – Zone names are always fully displayed (e.g. `std::sort`).
    - *Shortened* – Namespaces are shortened to one letter (e.g. `s::sort`).
    - *None* – Namespaces are completely omitted (e.g. `sort`).
- *Draw locks* – Controls the display of locks. If the *Only contended* option is selected, the non-blocking regions of locks won't be displayed (see section 5.2.3.3). The *Locks* drop-down allows disabling display of locks on a per-lock basis. As a convenience, the list of locks is split into the single-threaded and multi-threaded (contended and uncontended) categories. Clicking the right mouse button on a lock label opens the lock information window (section 5.16).
- *Draw plots* – Allows disabling display of plots. Individual plots can be disabled in the *Plots* drop-down.
- *Visible threads* – Here you can select which threads are visible on the timeline. Display order of threads can be changed by dragging thread labels.
- *Visible frame sets* – Frame set display can be enabled or disabled here. Note that disabled frame sets are still available for selection in the frame set selection drop-down (section 5.2.1), but are marked with a dimmed font.

Disabling display of some events is especially recommended when the profiler performance drops below acceptable levels for interactive usage.

5.4 Messages window

In this window you can see all the messages that were sent by the client application, as described in section 3.7. The window is split into four columns: *time*, *thread*, *message* and *call stack*. Hovering the mouse cursor over a message will highlight it on the timeline view. Clicking the left mouse button on a message will center the timeline view on the selected message.

The *call stack* column is filled only if a call stack capture was requested, as described in section 3.10. A single entry consists of the *Show* button, which opens the call stack information window (chapter 5.13) and of an abbreviated information about call path.

If the *Show frame images* option is selected, hovering the mouse cursor over a message will show a tooltip containing frame image (see section 3.3.3) associated with frame in which the message was issued, if available.

[52]There is an assumption that drift is linear. Automated measurement calculates and removes change over time in delay-to-execution of GPU zones. Resulting value may still be incorrect.

In a live capture, the message list will automatically scroll down to display the most recent message. This behavior can be disabled by manually scrolling the message list up. When the view is scrolled down to display the last message, the auto-scrolling feature will be enabled again.

The message list can be filtered in the following ways:

- By the originating thread in the *Visible threads* drop-down.
- By matching the message text to the expression in the *Filter messages* entry field. Multiple filter expressions can be comma-separated (e.g. 'warn, info' will match messages containing strings 'warn' *or* 'info'). Matches can be excluded by preceding the term with a minus character (e.g. '-debug' will hide all messages containing string 'debug').

5.5 Statistics window

Looking at the timeline view gives you a very localized outlook on things. Sometimes you want to take a look at the general overview of the program's behavior, for example you want to know which function takes the most of application's execution time. The statistics window provides you exactly that information.

If the trace capture was performed with call stack sampling enabled (as described in chapter 3.13.4), you will be presented with an option to switch between *Instrumentation* and *Sampling* modes. If no sampling data was collected, but symbols were retrieved, the second mode will be displayed as *Symbols*, enabling you to list available symbols. Otherwise only the instrumentation view will be present.

5.5.1 Instrumentation mode

Here you will find a multi-column display of captured zones, which contains: the zone *name* and *location*, *total time* spent in the zone, the *count* of zone executions and the *mean time spent in the zone per call*. The view may be sorted according to the three displayed values.

The *Self time* option determines how the displayed time is calculated. If it is disabled, the measurements will be inclusive, that is, containing execution time of zone's children. Enabling the option switches the measurement to exclusive, displaying just the time spent in zone, subtracting the child calls.

Clicking the left mouse button on a zone will open the individual zone statistics view in the find zone window (section 5.6).

You can filter the displayed list of zones by matching the zone name to the expression in the *Filter zones* entry field. Refer to section 5.4 for a more detailed description of the expression syntax.

5.5.2 Sampling mode

Data displayed in this mode is in essence very similar to the instrumentation one. Here you will find function names, their locations in source code and time measurements. There are, however, some very important differences.

First and foremost, the presented information is constructed from a number of call stack samples, which represent real addresses in the application's binary code, mapped to the line numbers in the source files. This reverse mapping may not be always possible, or may be erroneous. Furthermore, due to the nature of the sampling process, it is impossible to obtain exact time measurement. Instead, time values are guesstimated by multiplying number of sample counts by mean time between two distinct samples.

The *Name* column contains name of the function in which the sampling was done. If the *Inlines* option is enabled, functions which were inlined will be preceded with a '▶' symbol and additionally display their parent function name in parenthesis. Otherwise, only non-inlined functions are listed, with count of inlined functions in parenthesis. Any entry containing inlined function may be expanded to display the corresponding functions list (some functions may be hidden if the *Show all* option is disabled, due to lack of sampling data). Clicking on a function name will open the call stack sample parents window (see chapter 5.14). Note that if inclusive times are displayed, listed functions will be partially or completely coming from mid-stack frames, which will prevent, or limit the capability to display parent call stacks.

The *Location* column displays the corresponding source file name and line number. Depending on the *Location* option selection it can either show function entry address, or the instruction at which the sampling was performed. The *Entry* mode points at the beginning of a non-inlined function, or at the place where inlined function was inserted in its parent function. The *Sample* mode is not useful for non-inlined functions, as it points to one randomly selected sampling point out of many that were captured. However, in case of inlined functions, this random sampling point is within the inlined function body. Using these options in tandem enable you to look at both the inlined function code and the place where it was inserted. If the *Smart* location is selected, profiler will display entry point position for non-inlined functions and sample location for inlined functions. Selecting the *Address* option will instead print the symbol address.

The location data is complemented by the originating executable image name, contained in the *Image* column.

Some function locations may not be found, due to insufficient debugging data available on the client side. To filter out such entries, use the *Hide unknown* option.

The *Time* or *Count* column (depending on the *Show time* option selection) shows number of taken samples, either as a raw count, or in an easier to understand time format.

The last column, *Code size*, displays the size of symbol in the executable image of the program. Since inlined routines are directly embedded into other functions, their symbol size will be based on the parent symbol, and displayed as 'less than'. In some cases this data won't be available.

Finally, the list can be filtered using the *Filter symbols* entry field, just like in the instrumentation mode case, and the exclusive/inclusive time counting mode can be switched using the *Self time* switch. If the *Show all* option is selected, the list will include not only call stack samples, but also all other symbols collected during the profiling process (this is enabled by default, if no sampling was performed).

5.6 Find zone window

The individual behavior of zones may be influenced by many factors, like CPU cache effects, access times amortized by the disk cache, thread context switching, etc. Sometimes the execution time depends on the internal data structures and their response to different inputs. In other words, it is hard to determine the true performance characteristics by looking at any single zone.

Tracy gives you the ability to display an execution time histogram of all occurrences of a zone. On this view you can see how the function behaves in general. You can inspect how various data inputs influence the execution time and you can filter the data to eventually drill down to the individual zone calls, so that you can see the environment in which they were called.

You start by entering a search query, which will be matched against known zone names (see section 3.4 for information on the grouping of zone names). If the search found some results, you will be presented with a list of zones in the *matched source locations* drop-down. The selected zone's graph is displayed on the *histogram* drop-down and also the matching zones are highlighted on the timeline view. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.15).

An example histogram is presented on figure 21. Here you can see that the majority of zone calls (by count) are clustered in the 300 ns group, closely followed by the 10 µs cluster. There are some outliers at the 1 and 10 ms marks, which can be ignored on most occasions, as these are single occurrences.

The histogram is accompanied by various data statistics about displayed data, for example the *total time* of the displayed samples, or the *maximum number of counts* in histogram bins. The following options control how the data is presented:

- *Log values* – Switches between linear and logarithmic scale on the y axis of the graph, representing the call counts[53].
- *Log time* – Switches between linear and logarithmic scale on the x axis of the graph, representing the time bins.

[53]Or time, if the *cumulate time* option is enabled.

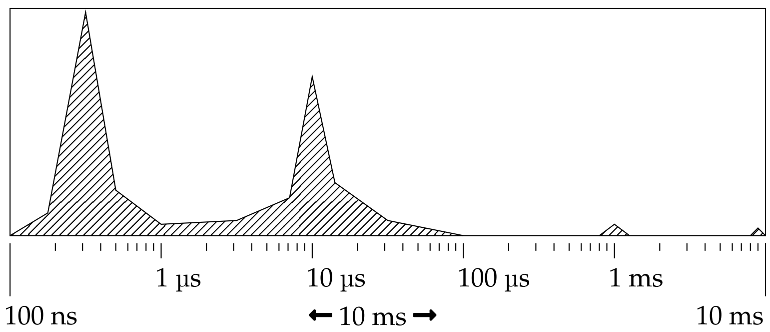

Figure 21: *Zone execution time histogram. Note that the extreme time labels and time range indicator (middle time value) are displayed in a separate line.*

- *Cumulate time* – Changes how the histogram bin values are calculated. By default the vertical bars on the graph represent the *call counts* of zones that fit in the given time bin. If this option is enabled, the bars represent the *time spent* in the zones. For example, on graph presented on figure 21 the 10 µs cluster is the dominating one, if we look at the time spent in zone, even if the 300 ns cluster has greater number of call counts.
- *Self time* – Removes children time from the analysed zones, which results in displaying only the time spent in the zone itself (or in non-instrumented function calls). Cannot be selected when *Running time* is active.
- *Running time* – Removes time when zone's thread execution was suspended by the operating system due to preemption by other threads, waiting for system resources, lock contention, etc. Available only when context switch capture was performed (section 3.13.2). Cannot be selected when *Self time* is active.
- *Minimum values in bin* – Excludes display of bins which do not hold enough values at both ends of the time range. Increasing this parameter will eliminate outliers, allowing to concentrate on the interesting part of the graph.

You can drag the left mouse button over the histogram to select a time range that you want to closely look at. This will display the data in the histogram info section and it will also filter zones displayed in the *found zones* section. This is quite useful, if you want to actually look at the outliers, i.e. where did they originate from, what the program was doing at the moment, etc[54]. You can reset the selection range by pressing the right mouse button on the histogram.

The *found zones* section displays the individual zones grouped according to the following criteria:

- *Thread* – In this mode you can see which threads were executing the zone.
- *User text* – Splits the zones according to the custom user text (see section 3.4).
- *Call stacks* – Zones are grouped by the originating call stack (see section 3.10). Note that two call stacks may sometimes appear identical, even if they are not, due to an easy to overlook difference in the source line numbers.
- *Parent* – Groups zones according to the parent zone. This mode relies on the zone hierarchy, and *not* on the call stack information.
- *No grouping* – Disables zone grouping. May be useful in cases when you just want to see zones in order as they appeared.

[54]More often than not you will find out, that the application was just starting, or an access to a cold file was required and there's not much you can do to optimize that particular case.

Each group may be sorted according to the *order* in which it appeared, the call *count*, the total *time* spent in the group, or the *mean time per call*. Expanding the group view will display individual occurrences of the zone, which can be sorted by application's time, execution time or zone's name. Clicking the left mouse button on a zone will open the zone information window (section 5.12). Clicking the middle mouse button on a zone will zoom the timeline view to the zone's extent.

Clicking the left mouse button on group name will highlight the group time data on the histogram (figure 22). This function provides a quick insight about the impact of the originating thread, or input data on the zone performance. Clicking the right mouse button on the group names area will reset the group selection.

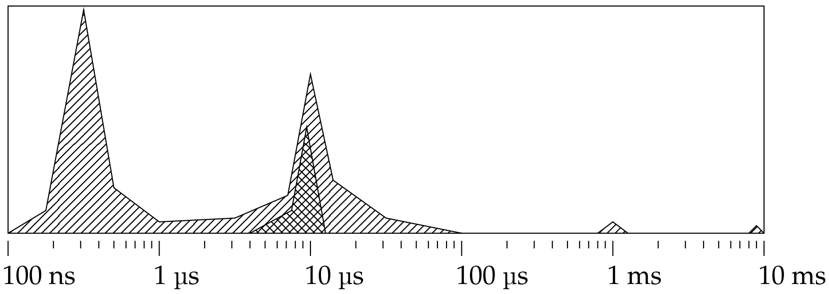

Figure 22: *Zone execution time histogram with a group highlighted.*

The call stack grouping mode has a different way of listing groups. Here only one group is displayed at any time, due to need to display the call stack frames. You can switch between call stack groups by using the ◀ and ▶ buttons. The group can be selected by clicking on the ✔ *Select* button (to reset the group selection use the right mouse button, as usual). You can open the call stack window (section 5.13) by pressing the ≣ *Call stack* button.

Tracy displays a variety of statistical values regarding the selected function: mean (average value), median (middle value), mode (most common value, quantized using histogram bins), and σ (standard deviation). The mean and median zone times are also displayed on the histogram as a red (mean) and blue (median) vertical bars. When a group is selected, additional bars will indicate the mean group time (orange) and median group time (green). You can disable drawing of either set of markers by clicking on the check-box next to the color legend.

Hovering the mouse cursor over a zone on the timeline, which is currently selected in the find zone window, will display a pulsing vertical bar on the histogram, highlighting the bin to which the hovered zone has been assigned. Zone entry on the zone list will also be highlighted.

Keyboard shortcut

You may press Ctrl + F to open or focus the find zone window and set the keyboard input on the search box.

Caveats

When using the execution times histogram you must be aware about the hardware peculiarities. Read section 2.2.2 for more detail.

5.6.1 Timeline interaction

When the zone statistics are displayed in the find zone menu, matching zones will be highlighted on the timeline display. Highlight colors match the histogram display. Bright blue highlight is used to indicate that a zone is in the optional selection range, while the yellow highlight is used for the rest of zones.

5.6.2 Frame time graph interaction

The frame time graph (section 5.2.2) behavior is altered when a zone is displayed in the find zone window and the *Show zone time in frames* option is selected. Instead of coloring the frame bars according to the frame time targets, an accumulated zone execution time is shown.

Each bar is drawn in gray color, with the white part accounting for the zone time. If the execution time is greater than the frame time (this is possible if more than one thread was executing the same zone), the overflow will be displayed using red color.

Enabling *Self time* option has an effect on the displayed values.

Caveats

The displayed data might not be calculated correctly and some zones may not be included in the reported times.

5.6.3 Limiting zone time range

If the *limit range* option is selected, only the zones within the specified time range will be included in the data. To indicate that the display is locked to a subset of all zones, a 🔒 lock icon will be displayed, and a yellow highlight matching the specified range will be present on the timeline view.

5.7 Compare traces window

Comparing the performance impact of the optimization work is not an easy thing to do. Benchmarking is often inconclusive, if even possible, in case of interactive applications, where the benchmarked function might not have a visible impact on frame render time. Doing isolated micro-benchmarks loses the execution environment of the application, in which many different functions compete for limited system resources.

Tracy solves this problem by providing a compare traces functionality, very similar to the find zone window, described in section 5.6. Traces can be compared either by zone or frame timing data.

You would begin your work by recording a reference trace that represents the usual behavior of the program. Then, after the optimization of the code is completed, you record another trace, doing roughly what you did for the reference one. Having the optimized trace open you select the 📂 *Open second trace* option in the compare traces window and load the reference trace.

Now things start to get familiar. You search for a zone, similarly like in the find zone window, choose the one you want in the *matched source locations* drop-down, and then you look at the histogram[55]. This time there are two overlaid graphs, one representing the current trace, and the second one representing the external (reference) trace (figure 23). You can easily see how the performance characteristics of the zone were affected by your modifications.

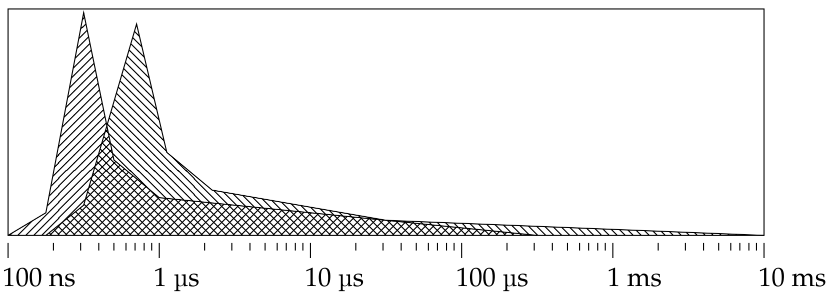

Figure 23: *Compare traces histogram.*

[55] When comparing frame times you are presented with a list of available frame sets, without the search box.

Note that the traces are color and symbol coded. The current trace is marked by a yellow symbol, and the external one is marked by a red symbol.

When searching for source locations it's not uncommon to match more than one zone (for example a search for `Draw` may result in `DrawCircle` and `DrawRectangle` matches). Typically you wouldn't want to compare execution profiles of two unrelated functions, which is prevented by the *link selection* option, which ensures that when you choose a source location in one trace, the corresponding one is also selected in second trace. Be aware that this may still result in a mismatch, for example if you have overloaded functions. In such case you will need to manually select the appropriate function in the other trace.

It may be difficult, if not impossible, to perform identical runs of a program. This means that the number of collected zones may differ in both traces, which would influence the displayed results. To fix this problem enable the *Normalize values* option, which will adjust the displayed results as-if both traces had the same number of recorded zones.

Trace descriptions

Set custom trace descriptions (see section 5.11) to easily differentiate the two loaded traces. If no trace description is set, a name of the profiled program will be displayed along with the capture time.

5.8 Memory window

The data gathered by profiling memory usage (section 3.8) can be viewed in the memory window. The top row contains statistics, such as *total allocations* count, number of *active allocations*, current *memory usage* and process *memory span*[56].

The lists of captured memory allocations are displayed in a common multi-column format thorough the profiler. The first column specifies the memory address of an allocation, or an address and an offset, if the address is not at the start of the allocation. Clicking the left mouse button on an address will open the memory allocation information window[57] (see section 5.10). Clicking the middle mouse button on an address will zoom the timeline view to memory allocation's range. The next column contains the allocation size.

The allocation's timing data is contained in two columns: *appeared at* and *duration*. Clicking the left mouse button on the first one will center the timeline view at the beginning of allocation, and likewise, clicking on the second one will center the timeline view at the end of allocation. Note that allocations that have not yet been freed will have their duration displayed in green color.

The memory event location in the code is displayed in the last four columns. The *thread* column contains the thread where the allocation was made and freed (if applicable), or an *alloc / free* pair of threads, if it was allocated in one thread and freed in another. The *zone alloc* contains the zone in which the allocation was performed[58], or - if there was no active zone in the given thread at the time of allocation. Clicking the left mouse button on the zone name will open the zone information window (section 5.12). Similarly, the *zone free* column displays the zone which freed the allocation, which may be colored yellow, if it is the same exact zone that did the allocation. Alternatively, if the zone has not yet been freed, a green *active* text is displayed. The last column contains the *alloc* and *free* call stack buttons, or their placeholders, if no call stack is available (see section 3.10 for more information). Clicking on either of the buttons will open the call stack window (section 5.13). Note that the call stack buttons that match the information window will be highlighted.

The memory window is split into the following sections:

[56]Memory span describes the address space consumed by the program. It is calculated as a difference between the maximum and minimum observed in-use memory address.

[57]While the allocation information window is opened, the address will be highlighted on the list.

[58]The actual allocation is typically a couple functions deeper in the call stack.

5.8.1 Allocations

The @ *Allocations* pane allows you to search for the specified address usage during the whole life-time of the program. All recorded memory allocations that match the query will be displayed on a list.

5.8.2 Active allocations

The *Active allocations* pane displays a list of currently active memory allocations and their total memory usage. Here you can see where exactly your program did allocate memory it is currently using. If the application has already exited, this becomes a list of leaked memory.

5.8.3 Memory map

On the *Memory map* pane you can see the graphical representation of your program's address space. Active allocations are displayed as green lines, while the freed memory is marked as red lines. The brightness of the color indicates how much time has passed since the last memory event at the given location – the most recent events are the most vibrant.

This view may be helpful in assessing the general memory behavior of the application, or in debugging the problems resulting from address space fragmentation.

5.8.4 Bottom-up call stack tree

The *Bottom-up call stack tree* pane is only available, if the memory events were collecting the call stack data (section 3.10). In this view you are presented with a tree of memory allocations, starting at the call stack entry point and going up to the allocation's pinpointed place. Each level of the tree is sorted according to the number of bytes allocated in given branch.

Each tree node consists of three elements: the function name, the source file location and the memory allocation data. The memory allocation data is either yellow *inclusive* events count (allocations performed by children), or the cyan *exclusive* events count (allocations that took place in the node)[59]. There are two values that are counted: total memory size and number of allocations.

The *Group by function name* option controls how tree nodes are grouped. If it is disabled, then the grouping is performed at a machine instruction level granularity. This may result in very verbose output, but the displayed source locations are precise. To make the tree more readable you may opt to perform grouping at the function name level, which will result in less valid source file locations, as multiple entries are collapsed into one.

Enabling the *Only active allocations* option will limit the call stack tree to only display active allocations.

Clicking the right mouse button on the function name will open allocations list window (see section 5.9), which list all the allocations included at the current call stack tree level. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.15).

Some function names may be too long to be properly displayed, with the events count data at the end. In such cases, you may press the *control* button, which will display events count tooltip.

5.8.5 Top-down call stack tree

This pane is identical in functionality to the *Bottom-up call stack tree*, but the call stack order is reversed when the tree is built. This means that the tree starts at the memory allocation functions and goes down to the call stack entry point.

[59]Due to the way call stacks work there is no possibility for an entry to have both inclusive and exclusive counts, in a properly instrumented program.

5.8.6 Looking back at the memory history

By default the memory window displays the memory data at the current point of program execution. It is however possible to view the historical data by enabling the *Restrict time* option. This will draw a vertical violet line on the timeline view, which will act as a terminator for memory events. The memory window will use only the events lying on the left side of the terminator line (in the past), ignoring everything that's on the right side.

5.9 Allocations list window

This window displays the list of allocations included at the selected call stack tree level (see section 5.8 and 5.8.4).

5.10 Memory allocation information window

The information about the selected memory allocation is displayed in this window. It lists the allocation's address and size, along with the time, thread and zone data of the allocation and free events. Clicking the *Zoom to allocation* button will zoom the timeline view to the allocation's extent.

5.11 Trace information window

This window contains information about the current trace: captured program name, time of the capture, profiler version which performed the capture and a custom trace description, which you can fill in.

Open the *Trace statistics* section to see information about the trace, such as achieved timer resolution, number of captured zones, lock events, plot data points, memory allocations, etc.

There's also a section containing the selected frame set timing statistics and histogram[60]. As a convenience you can switch the active frame set here and limit the displayed frame statistics to the frame range visible on the screen.

If *CPU topology* data is available (see section 3.13.3), you will be able to view the package, core and thread hierarchy.

The *Source location substitutions* section allows adapting the source file paths, as captured by the profiler to the actual on-disk locations[61]. You can create a new substitution by clicking the *Add new substitution* button. This will add a new entry, with input fields for ECMAScript-conforming regular expression pattern and its corresponding replacement string. The outcome of substitutions can be quickly tested in the *example source location* input field, which will be transformed and displayed below, as *result*.

Quick example

Let's say we have an unix-based operating system with program sources in `/home/user/program/src/` directory. We have also performed a capture of an application running under Windows, with sources in `C:\Users\user\Desktop\program\src` directory. Obviously, the source locations don't match and the profiler can't access the source files we have on our disk. We can fix that by adding two substitution patterns:

- `^C:\\Users\\user\\Desktop` → `/home/user`
- `\\` → `/`

[60] See section 5.6 for a description of the histogram. Note that there are subtle differences in the available functionality.
[61] This has no effect on source files cached during the profiling run.

In this window you can view the information about the machine on which the profiled application was running. This includes the operating system, used compiler, CPU name, amount of total available RAM, etc. If application information was provided (see section 3.7.1), it will also be displayed here.

If an application should crash during profiling (section 2.5), the crash information will be displayed in this window. It provides you information about the thread that has crashed, the crash reason and the crash call stack (section 5.13).

5.12 Zone information window

The zone information window displays detailed information about a single zone. There can be only one zone information window open at any time. While the window is open the zone will be highlighted on the timeline view with a green outline. The following data is presented:

- Basic source location information: function name, source file location and the thread name.
- Timing information.
- If context switch capture was performed (section 3.13.2) and a thread was suspended during zone execution, a list of wait regions will be displayed, with complete information about timing, CPU migrations and wait reasons. If CPU topology data is available (section 3.13.3), zone migrations across cores will be marked with 'C', and migrations across packages – with 'P'. In some cases context switch data might be incomplete[62], in which case a warning message will be displayed.
- Memory events list, both summarized and a list of individual allocation/free events (see section 5.8 for more information on the memory events list).
- List of messages that were logged in the zone's scope (including its children).
- Zone trace, taking into account the zone tree and call stack information (section 3.10), trying to reconstruct a combined zone + call stack trace[63]. Captured zones are displayed as normal text, while functions that were not instrumented are dimmed. Hovering the mouse pointer over a zone will highlight it on the timeline view with a red outline. Clicking the left mouse button on a zone will switch the zone info window to that zone. Clicking the middle mouse button on a zone will zoom the timeline view to the zone's extent. Clicking the right mouse button on a source file location will open the source file view window (if applicable, see section 5.15).
- Child zones list, showing how the current zone's execution time was used. Zones on this list can be grouped according to their source location. Each group can be expanded to show individual entries. All the controls from the zone trace are also available here.
- Time distribution in child zones, which expands the information provided in the child zones list by processing *all* zone children (including multiple levels of grandchildren). This results in a statistical list of zones that were really doing the work in the current zone's time span. If a group of zones is selected on this list, the find zone window (section 5.6) will open, with time range limited to show only the children of the current zone.

The zone information window has the following controls available:

- *Zoom to zone* – Zooms the timeline view to the zone's extent.
- *Go to parent* – Switches the zone information window to display current zone's parent zone (if available).

[62]For example, when a capture is ongoing and context switch information has not yet been received.

[63]Reconstruction is only possible, if all zones have full call stack capture data available. In case where that's not available, an *unknown frames* entry will be present.

- *Statistics* – Displays the zone general performance characteristics in the find zone window (section 5.6).
- *Call stack* – Views the current zone's call stack in the call stack window (section 5.13). The button will be highlighted, if the call stack window shows the zone's call stack. Only available if zone had captured call stack data (section 3.10).
- *Source* – Display source file view window with the zone source code (only available if applicable, see section 5.15). Button will be highlighted, if the source file is being currently displayed (but the focused source line might be different).
- *Go back* – Returns to the previously viewed zone. The viewing history is lost when the zone information window is closed, or when the type of displayed zone changes (from CPU to GPU or vice versa).

5.13 Call stack window

This window shows the frames contained in the selected call stack. Each frame is described by a function name, source file location and originating image[64] name. Clicking the left mouse button on either the function name of source file location will copy the name to the clipboard. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.15).

A single stack frame may have multiple function call places associated with it. This happens in case of inlined function calls. Such entries will be displayed in the call stack window, with *inline* in place of frame number[65].

Stack frame location may be displayed in the following number of ways, depending on the *Frame location* option selection:

- *Source code* – displays source file and line number associated with the frame.
- *Entry point* – source code at the beginning of the function containing selected frame, or function call place in case of inline frames.
- *Return address* – shows return address, which may be used to pinpoint the exact instruction in the disassembly.
- *Symbol address* – displays begin address of the function containing the frame address.

In some cases it may be not possible to properly decode stack frame address. Such frames will be presented with a dimmed '`[ntdll.dll]`' name of the image containing the frame address, or simply '`[unknown]`' if even this information cannot be retrieved. Additionally, '`[kernel]`' is used to indicate unknown stack frames within the operating system internal routines.

If the displayed call stack is a sampled call stack (chapter 3.13.4), an additional button will be available, *Global entry statistics*. Clicking it will open the call stack sample parents window (chapter 5.14) for the current call stack.

5.13.1 Reading call stacks

You need to take special care when reading call stacks. Contrary to their name, call stacks do not show *function call stacks*, but rather *function return stacks*. This might be a bit confusing at first, but this is how programs do work. Consider the following source code:

```
int main()
{
    auto app = std::make_unique<Application>();
```

[64]Executable images are called *modules* by Microsoft.
[65]Or '▸' icon in case of call stack tooltips.

```
        app->Run();
        app.reset();
    }
```

Let's say you are looking at the call stack of some function called within `Application::Run`. This is the result you might get:

```
0. ...
1. ...
2. Application::Run
3. std::unique_ptr<Application>::reset
4. main
```

At the first glance it may look like `unique_ptr::reset` was the *call site* of the `Application::Run`, which would make no sense, but this is not the case here. When you remember these are the *function return points*, it becomes much more clear what is happening. As an optimization, `Application::Run` is returning directly into `unique_ptr::reset`, skipping the return to `main` and an unnecessary `reset` function call.

Moreover, the linker may determine in some rare cases that any two functions in your program are identical[66]. As a result, only one copy of the binary code will be provided in the executable for both functions to share. While this optimization produces more compact programs, it also means that there's no way to distinguish the two functions apart in the resulting machine code. In effect, some call stacks may look nonsensical until you perform a small investigation.

5.14 Call stack sample parents window

This window displays statistical information about the selected symbol. All sampled call stacks (chapter 3.13.4) leading to the symbol are counted and displayed in descending order. You can select the displayed call stack using the *parent call stack* controls, which also display time spent in the chosen call stack. Alternatively, sample counts may be shown by disabling the ⏱ *Show time* option, which is described in more detail in chapter 5.5.2.

The layout of frame list and the @ *Frame location* option selection is similar to the call stack window, described in chapter 5.13.

5.15 Source file view window

In this window you can view the source code of the profiled application, to take a quick glance at the context of the function behavior you are analyzing. The selected line (for example, a location of a profiling zone) will be highlighted both in the source code listing and on the scroll bar.

> **Important**
>
> Source file view depends on local files you have on your disk, as the profiled application doesn't need them to run. If the profiler can access the source files during capture, it will cache them for further reference. Otherwise, you will need to make them available, possibly by using file path substitutions. Keep the following rules in mind:
>
> - Source files can only be used, if the source file location recorded in the trace matches the files you have on your disk. See section 5.11 for information on redirecting source file locations.
> - Time stamp of the source file cannot be newer than the trace, as it typically would indicate that the file has been changed and no longer contains the code that was profiled.

[66] For example if all they do is zero-initialize a region of memory. As some constructors would do.

> • **The displayed source files might not reflect the code that was profiled!** It is up to you to verify that you don't have a modified version of the code, with regards to the trace.

5.15.1 Symbol view

If the inspected source location has an associated symbol context (i.e. if it comes from a call stack capture, from call stack sampling, etc.), a much more capable symbol view is available. A symbol is an unit of machine code, basically a callable function. It may be generated using multiple source files and may consist of multiple inlined functions. A list of all captured symbols is available in the statistics window, as described in chapter 5.5.2.

The header of symbol view window contains a name of the selected *symbol*, a list of *functions* that contribute to the symbol, and information such as *Code size* in the program, or count of probed *Samples*.

Additionally, you may use the *Mode* selector to decide what content should be displayed in the panels below:

- *Source* – only the source code will be displayed.
- *Assembly* – only the machine code disassembly will be shown.
- *Combined* – both source code and disassembly will be listed next to each other.

In some circumstances (missing or outdated source files, lack of machine code) some modes may be unavailable.

5.15.1.1 Source mode

This is pretty much the original source file view window, but with the ability to select one of the source files that were used to build the symbol. Additionally, each source file line which produced machine code in the symbol will show count of associated assembly instructions, displayed with an '`@`' prefix, and will be marked with a grey color on the scroll bar. Due to the way optimizing compilers work, some lines may seemingly not produce any machine code, for example because iterating a loop counter index might have been reduced to advancing a data pointer. Some other lines may have disproportionate amount of associated instructions, e.g. when a loop unrolling optimization is applied. This varies from case to case and from compiler to compiler.

5.15.1.2 Assembly mode

This mode shows the disassembly of the symbol machine code. Each assembly instruction is displayed listed with its location in the program memory during execution. If the *Relative locations* option is selected, an offset from the symbol beginning will be printed instead. Clicking the left mouse button on the address/offset will switch to counting line numbers, using the selected one as origin (i.e. zero value). Line numbers are displayed inside `[]` brackets. This display mode can be useful to correlate lines with output of external tools, such as `llvm-mca`. To disable line numbering click the right mouse button on a line number.

If the *Source locations* option is selected, each line of the assembly code will also contain information about the originating source file name and line number. For easier differentiation between different source files, each file is assigned its own color. Clicking the left mouse button on a displayed source location will switch the source file, if necessary, and focus the source view on selected line.

Selecting the *Machine code* option will enable display of raw machine code bytes for each line.

If any instruction would jump to a predefined address, symbolic name of the jump target will be additionally displayed. If the destination location is within the currently displayed symbol an -> arrow will be prepended to the name. Hovering the mouse pointer over such symbol name will highlight the target location. Clicking on it with the left mouse button will focus the view on the destination instruction, or switch view to the destination symbol.

Enabling the *Jumps* option will show jumps within the symbol code as a series of arrows from the jump source to the jump target. Hovering the mouse pointer over a jump arrow will display jump information tooltip and will also draw the jump range on the scroll bar, as a green line. Jump target location will be marked by a horizontal green line. Clicking on a jump arrow with the left mouse button will focus the view on the target location. Jumps going out of the symbol[67] will be indicated by a smaller arrow pointing away from the code.

The *AT&T* switch can be used to select between *Intel* and *AT&T* assembly syntax. Beware that microarchitecture data is only available if Intel syntax is selected.

Unlike the source file view, portions of the executable are stored within the captured profile and don't rely on the local disk files being available.

Exploring microarchitecture If the listed assembly code targets x86 or x64 instruction set architectures, hovering mouse pointer over an instruction will display a tooltip with microarchitectural data, based on measurements made in [AR19]. *This information is retrieved from instruction cycle tables, and does not represent true behavior of the profiled code.* Reading the cited article will give you a detailed definition of the presented data, but here's a quick (and inaccurate) explanation:

- *Throughput* – How many cycles are required to execute an instruction in a stream of independent same instructions. For example, if two independent `add` instructions may be executed simultaneously on different execution units, then the throughput (cycle cost per instruction) is 0.5.
- *Latency* – How many cycles it takes for an instruction to finish executing. This is reported as a min-max range, as some output values may be available earlier than the rest.
- *μops* – How many microcode operations have to be dispatched for an instruction to retire. For example, adding a value from memory to a register may consist of two microinstructions: first load the value from memory, then add it to the register.
- *Ports* – Which ports (execution units) are required for dispatch of microinstructions. For example, `2*p0+1*p015` would mean that out of the three microinstructions implementing the assembly instruction, two can only be executed on port 0, and one microinstruction can be executed on ports 0, 1, or 5. Number of available ports and their capabilities vary between different processors architectures. Refer to `https://wikichip.org/` for more information.

Selection of the CPU microarchitecture can be performed using the *μarch* drop-down. Each architecture is accompanied with a name of an example CPU implementing it.

Enabling the *Latency* option will display graphical representation of instruction latencies on the listing. Minimum latency of an instruction is represented with a red bar, while the maximum latency is represented by a yellow bar.

Clicking on the *Save* button lets you write the disassembly listing to a file. You can then manually extract some critical loop kernel and pass it to a CPU simulator, such as *LLVM Machine Code Analyzer* (`llvm-mca`)[68], in order to see how the code is executed and if there are any pipeline bubbles. Consult the `llvm-mca` documentation for more details. Alternatively, you might click the right mouse button on a jump arrow and save only the instructions within the jump range, using the *Save jump range* button.

Instruction dependencies Assembly instructions may read values stored in registers and may also write values to registers. A dependency between two instructions is created when one produces some result, which is then consumed by the other one. Combining this dependency graph with information about instruction latencies may give deep understanding of the bottlenecks in code performance.

[67]This includes jumps, procedure calls and returns. For example, in x86 assembly the respective operand names can be: `jmp`, `call`, `ret`.

[68]`https://llvm.org/docs/CommandGuide/llvm-mca.html`

Clicking the left mouse button on any assembly instruction will mark it as a target for resolving register dependencies between instructions. To cancel this selection, click on any assembly instruction with right mouse button.

The selected instruction will be highlighted in red, while its dependencies will be highlighted in violet. Additionally, a list of dependent registers will be listed next to each instruction which reads or writes to them, with the following color code:

- *Green* – Register value is read (is a dependency *after* target instruction).
- *Red* – A value is written to a register (is a dependency *before* target instruction).
- *Yellow* – Register is read and then modified.
- *Grey* – Value in a register is either discarded (overwritten), or was already consumed by an earlier instruction (i.e. it is readily available[69]). Dependency will be not followed further.

Search for dependencies follows program control flow, so there may be multiple producers and consumers for any single register. While the *after* and *before* guidelines mentioned above hold in general case, things may be more complicated when there's a large amount of conditional jumps in the code. Note that dependencies further away than 64 instructions are not displayed.

For easier navigation, dependencies are also marked on the left side of the scroll bar, following the green, red and yellow convention. The selected instruction is marked in blue.

5.15.1.3 Combined mode

In this mode both the source and assembly panes will be displayed together, providing the best way to gain insight into the code. Hovering the mouse pointer over the source file line, or the location of the assembly line will highlight the corresponding lines in the second pane (both in the listing and on the scroll bar). Clicking the left mouse button on a line will select it in both panes. Clicking the right mouse button will also focus the secondary view on the selected line (or first of the selected lines, if more than one).

Important

An assembly instruction may be associated with only a single source line, but a source line might be associated with multiple assembly lines, sometimes intermixed with other assembly instructions.

5.15.1.4 Instruction pointer statistics

If automated call stack sampling (see chapter 3.13.4) was performed, additional profiling information will be available. The first column of source and assembly views will contain percentage counts of collected instruction pointer samples for each displayed line, both in numerical and graphical bar form. This information can be used to determine which line of the function takes the most time. The displayed percentage values are heat map color coded, with the lowest values mapped to dark red, and the highest values mapped to bright yellow. The color code will appear next to the percentage value, and on the scroll bar, so that 'hot' places in code can be identified at a glance.

Instruction timings can be viewed as a group. To begin constructing such group, click the left mouse button on the percentage value. Additional instructions can be added using the Ctrl key, while holding the ⇧ key will allow selection of a range. To cancel the selection, click the right mouse button on a percentage value. Group statistics can be seen at the bottom of the pane.

Clicking the middle mouse button on the percentage value of an assembly instruction will display call stack parents of the selected sample (see chapter 5.14). This functionality is only available for instructions

[69]This is actually a bit of simplification. Run a pipeline simulator, e.g. `llvm-mca` for a better analysis.

that have collected sampling data, and only in the assembly view, as the source code may be inlined multiple times, which would result in ambiguous location data.

Sample data source is controlled by the *Function* control, in the window header. If this option is disabled, the sample data represents the whole symbol. If it is enabled, then the sample data will only include the selected function.

> **Important**
>
> Be aware that the data is not fully accurate, as it is the result of random sampling of program execution. Furthermore, undocumented implementation details of an out-of-order CPU architecture will highly impact the measurement. Read chapter 2.2.2 to see the tip of an iceberg.

5.16 Lock information window

This window presents information and statistics about a lock. The lock events count represents the total number collected of wait, obtain and release events. The announce, termination and lock lifetime measure the time from the lockable construction until destruction.

5.17 Frame image playback window

You may view a live replay of the profiled application screen captures (see section 3.3.3) using this window. Playback is controlled by the *Play* and *Pause* buttons and the *Frame image* slider can be used to scrub to the desired time stamp. Alternatively you may use the and buttons to change single frame back or forward.

If the *Sync timeline* option is selected, the timeline view will be focused on the frame corresponding to the currently displayed screen shot. The *Zoom 2×* option enlarges the image, for easier viewing.

Each displayed frame image is also accompanied by the following parameters: *timestamp*, showing at which time the image was captured, *frame*, displaying the numerical value of corresponding frame, and *ratio*, telling how well the in-memory loss-less compression was able to reduce the image data size.

5.18 CPU data window

Statistical data about all processes running on the system during the capture is available in this window, if context switch capture (section 3.13.2) was performed.

Each running program has an assigned process identifier (PID), which is displayed in the first column. If a program entry is expanded, a list of thread identifiers (TIDs) will also be displayed.

The *running time* column shows how much processor time was used by a process or thread. The percentage may be over 100%, as it is scaled to trace length and multiple threads belonging to a single program may be executing simultaneously. The *running regions* column displays how many times a given entry was in the *running* state and the *CPU migrations* shows how many times an entry was moved from one CPU core to another, when an entry was suspended by the system scheduler.

The profiled program is highlighted using green color. Furthermore, yellow highlight indicates threads which are known to the profiler (that is, which sent events due to instrumentation).

5.19 Annotation settings window

In this window you may modify how a timeline annotation (section 5.2.5) is presented by setting its text description, or selecting region highlight color. If the note is no longer needed, it may also be removed here.

5.20 Annotation list window

This window lists all annotations marked on the timeline. Each annotation is presented, as shown on figure 24. From left to right the elements are:

- *Edit* – Opens the annotation settings window (section 5.19).
- *Zoom* – Zooms timeline to the annotation extent.
- *Remove* – Removes the annotation. You must press the Ctrl key to enable this button.
- Colored box – Color of the annotation.
- Text description of the annotation.

Figure 24: *Annotation list entry*

6 Importing external profiling data

Tracy can import data generated by other profilers. This external data cannot be directly loaded, but must be converted first. Currently there's only support for converting chrome:tracing data, through the `import-chrome` utility.

Limitations

- Tracy is a single-process profiler. There is no differentiation between data coming from different pids.
- Tracy uses thread identifiers assigned by the operating system. This means that no two concurrent threads can have the same tid. Be aware that some external data formats may encourage usage of duplicated thread identifiers.
- The imported data may be severely limited, either by not mapping directly to the data structures used by Tracy, or by following undocumented practices.

7 Configuration files

While the client part doesn't read or write anything to the disk (with the exception of accessing the `/proc` filesystem on Linux), the server part has to keep some persistent state. The naming conventions or internal data format of the files are not meant to be known by profiler users, but you may want to do a backup of the configuration, or move it to another machine.

On Windows settings are stored in the `%APPDATA%/tracy` directory. All other platforms use the `$XDG_CONFIG_HOME/tracy` directory, or `$HOME/.config/tracy` if the `XDG_CONFIG_HOME` environment variable is not set.

7.1 Root directory

Various files at the root configuration directory store common profiler state such as UI windows position, connections history, etc.

7.2 Trace specific settings

Trace files saved on disk are immutable and can't be changed, but it may be desirable to store additional per-trace information to be used by the profiler, for example a custom description of the trace, or the timeline view position used in the previous profiling session.

This external data is stored in the `user/[letter]/[program]/[week]/[epoch]` directory, relative to the configuration's root directory. The `program` part is the name of the profiled application (for example `program.exe`). The `letter` part is a first letter of the profiled application's name. The `week` part is a number of weeks since the unix epoch, and the `epoch` part is a number of seconds since unix epoch. This rather unusual convention prevents creation of directories with hundreds of entries.

User settings are never pruned by the profiler.

Appendices

A License

B List of contributors

```
Bartosz Taudul <wolf.pld@gmail.com>
Kamil Klimek <kamil.klimek@sharkbits.com>         (initial find zone implementation)
Bartosz Szreder <zgredder@gmail.com>              (view/worker split)
Arvid Gerstmann <dev@arvid-g.de>                  (compatibility fixes)
Rokas Kupstys <rokups@zoho.com>                   (compatibility fixes, initial CI work, MingW suppo
Till Rathmann <till.rathmann@gmx.de>              (DLL support)
Sherief Farouk <sherief.personal@gmail.com>       (compatibility fixes)
Dedmen Miller <dedmen@dedmen.de>                  (find zone bug fixes, improvements)
Michał Cichoń <michcic@gmail.com>                 (OSX call stack decoding backport)
Thales Sabino <thales@codeplay.com>               (OpenCL support)
Andrew Depke <andrewdepke@gmail.com>              (Direct3D 12 support)
```

C Inventory of external libraries

The following libraries are included with and used by the Tracy Profiler. Entries marked with a ★ icon are used in the client code.

- 3-clause BSD license
  - getopt_port – `https://github.com/kimgr/getopt_port`
  - libbacktrace ★ – `https://github.com/ianlancetaylor/libbacktrace`
  - Zstandard – `https://github.com/facebook/zstd`
  - capstone – `https://github.com/aquynh/capstone`
- 2-clause BSD license
  - concurrentqueue ★ – `https://github.com/cameron314/concurrentqueue`
  - LZ4 ★ – `https://github.com/lz4/lz4`
  - xxHash – `https://github.com/Cyan4973/xxHash`
- Public domain
  - rpmalloc ★ – `https://github.com/rampantpixels/rpmalloc`
  - gl3w – `https://github.com/skaslev/gl3w`
  - stb_image – `https://github.com/nothings/stb`
- zlib license
  - Native File Dialog – `https://github.com/mlabbe/nativefiledialog`
  - GLFW – `https://github.com/glfw/glfw`
  - IconFontCppHeaders – `https://github.com/juliettef/IconFontCppHeaders`
  - pdqsort – `https://github.com/orlp/pdqsort`
- MIT license
  - Dear ImGui – `https://github.com/ocornut/imgui`
  - JSON for Modern C++ – `https://github.com/nlohmann/json`
  - robin-hood-hashing – `https://github.com/martinus/robin-hood-hashing`
- Apache license 2.0
  - Arimo font – `https://fonts.google.com/specimen/Arimo`
  - Cousine font – `https://fonts.google.com/specimen/Cousine`
- Font Awesome Free License
  - Font Awesome – `https://fontawesome.com/`
- FreeType License
  - FreeType – `https://www.freetype.org/`

References

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.