CPPP 2021 PARIS

Performance is not (only) about micro-optimizations!

Clément Grégoire

MUREX

CODE RECKONS

Science to the CORE

Performance is not (only) about micro-optimizations!

Who am I?

Clément GREGOIRE

- 10 years of C++ Programming
- 5 years in game-dev
    - Microsoft Flight Simulator
    - Various ports
- Software optimization is my job
- Co-founder & consultant

This talk is

- Not meant to convince you performance is important
- Not your usual micro-benchmarking talk (obviously)
- Not specific to C++

"Performance Matters" by Emery Berger
https://youtu.be/r-TLSBdHe1A

siliceum

Software performance

What’s that?

Speed

How long does it take for a task to execute?
Latency?

- CPU
- GPU
- Network
- Startup / loading time

Memory usage

How much memory can we afford to use?

Do we need to play nice with other applications?

- Embedded
- Cloud
- Video games
  - PC vs Console
- Multi-tasking

Storage

What is my package size?

How long will it take to download?

- Embedded
- Docker images
- Games
  - Textures, sounds, 3D models, …
- Web
  - Images, minification, …

“Fallout 76’s Day One Patch is Larger Than the Actual Game”

DOWNLOAD POWERPOINT (1.5 GB)

Resilience

- Security (DoS)
- Scaling up
- Availability

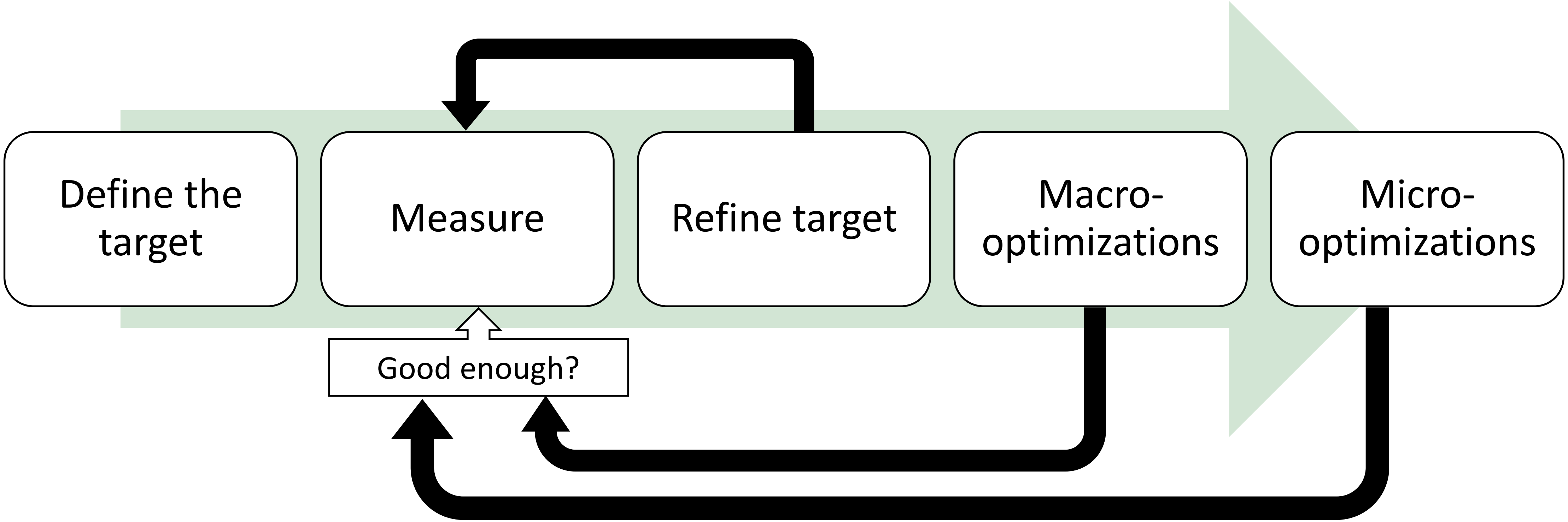
Maintainability

- Speed
  - Development / prototyping
  - Identification of performance issues
- Iteration/build time (Live++ anyone?)
- Debug mode performance

Optimizations

Where to begin?

The optimization cycle



Defining performance objectives

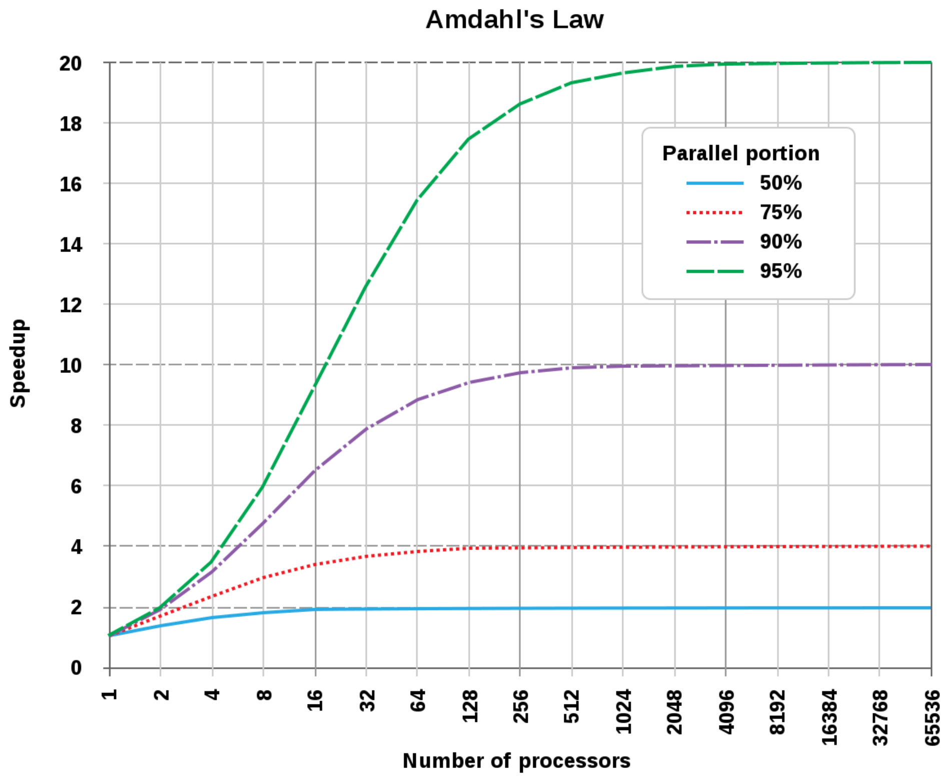
A few examples

- 60 frames per second
- 8GB of memory maximum
- Web page loading < 1s
- Be the best (=> no target, only if you have time to afford it)

Defining performance objectives

Feasibility and cost-effectiveness

- Can this actually be done?
- How long will it take?
- Is it an isolated use-case?
- Buy better hardware?
  (yes but **no**, end of Moore’s law, Amdahl’s law, …)



siliceum

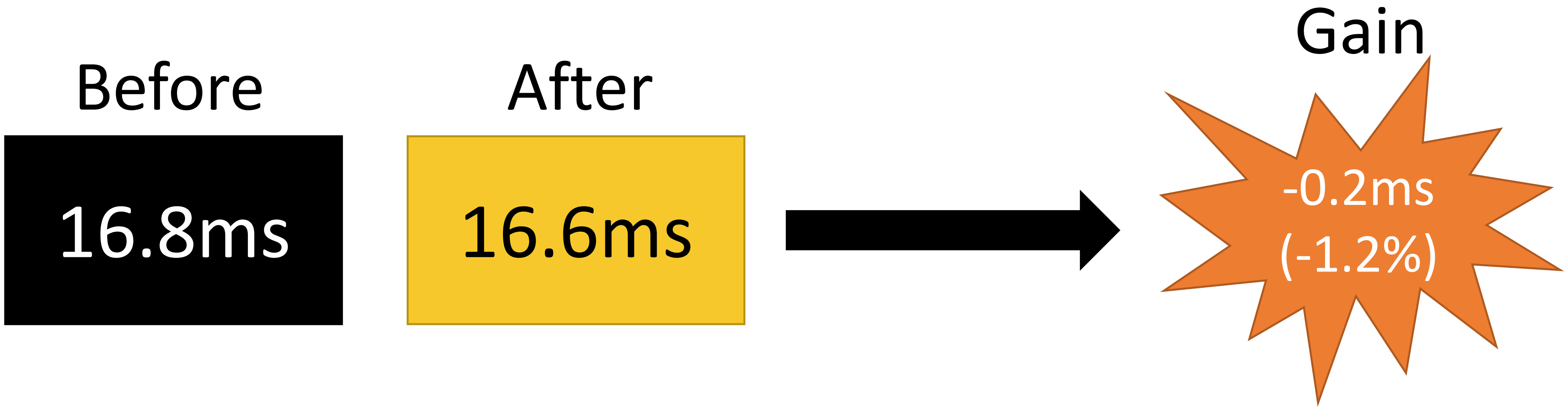
Measure

The most important part!

Performance testing

- Reproducible
- As deterministic as possible
- Choice of input data is important
  - Size must be realistic
  - Represents the system in production or at its limits

With a single measurement



Before

16.8ms

After

16.6ms

Gain

-0.2ms
(-1.2%)

Ship it?

Environment stability

(Do not leave YouTube running in the background)

- Close unnecessary programs
- Lock the CPU / GPU frequency
- Be wary of temperature and power supply
- Link order, environment variables, ASLR, code/allocations alignment…
- So much more

Sometimes from single to double

https://doc.calcite.siliceum.com/performance/stability

The solution

Repeat the experiment!

Use units correctly

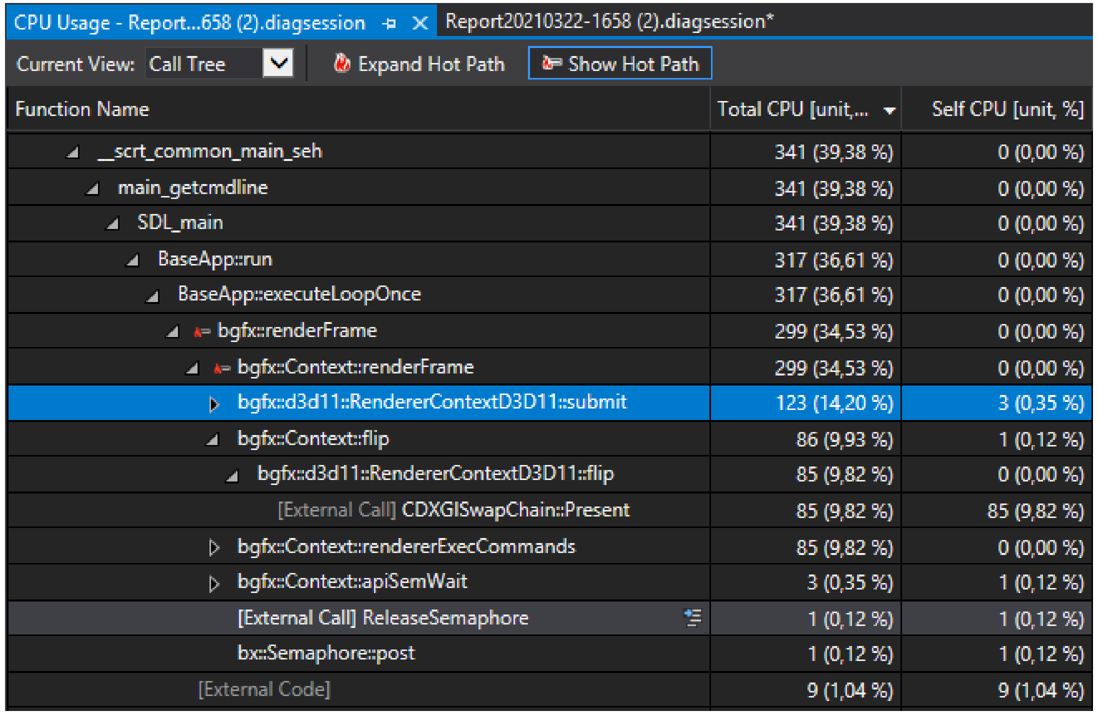
- % only if no other way to describe it correctly
  - Cache miss rate ☑
  - Branch misprediction ☑
  - Timings ✕
  - Memory ✕
- No unit expressed in « ops / X » (FPS, throughput)
  - Ok for a marketing effect
  - Bad when looking at improvements
  - Same modifications in different order => different results
- NO AVERAGE → Use median / percentiles / statistical tests…

siliceum

Profilers

Tools of the trade

Sampling-based profilers


CPU Usage - Report...658 (2).diagsession
Report20210322-1658 (2).diagsession*
Expand Hot Path
Show Hot Path

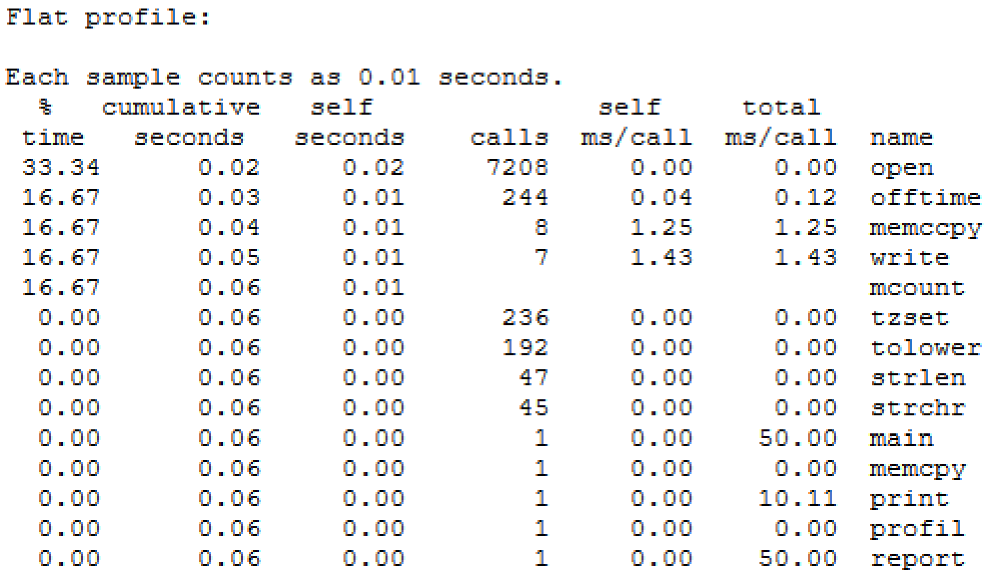
Current View: Call Tree

Function Name	Total CPU [unit,...	Self CPU [unit, %]
__scrt_common_main_seh	341 (39,38 %)	0 (0,00 %)
main_getcmdline	341 (39,38 %)	0 (0,00 %)
SDL_main	341 (39,38 %)	0 (0,00 %)
BaseApp::run	317 (36,61 %)	0 (0,00 %)
BaseApp::executeLoopOnce	317 (36,61 %)	0 (0,00 %)
bgfx::renderFrame	299 (34,53 %)	0 (0,00 %)
bgfx::Context::renderFrame	299 (34,53 %)	0 (0,00 %)
bgfx::d3d11::RendererContextD3D11::submit	123 (14,20 %)	3 (0,35 %)
bgfx::Context::flip	86 (9,93 %)	1 (0,12 %)
bgfx::d3d11::RendererContextD3D11::flip	85 (9,82 %)	0 (0,00 %)
[External Call] CDXGISwapChain::Present	85 (9,82 %)	85 (9,82 %)
bgfx::Context::rendererExecCommands	85 (9,82 %)	0 (0,00 %)
bgfx::Context::apiSemWait	3 (0,35 %)	1 (0,12 %)
[External Call] ReleaseSemaphore	1 (0,12 %)	1 (0,12 %)
bx::Semaphore::post	1 (0,12 %)	1 (0,12 %)
[External Code]	9 (1,04 %)	9 (1,04 %)

- Quick results ☑
- I/O not always sampled ⚠
- Variable precision ✕
- Time spent in function ≠Useful time in function ✕
- Linux Perf, Visual Studio, Intel VTune

Automatic instrumentation

(Avoid it 99,999% of the time)

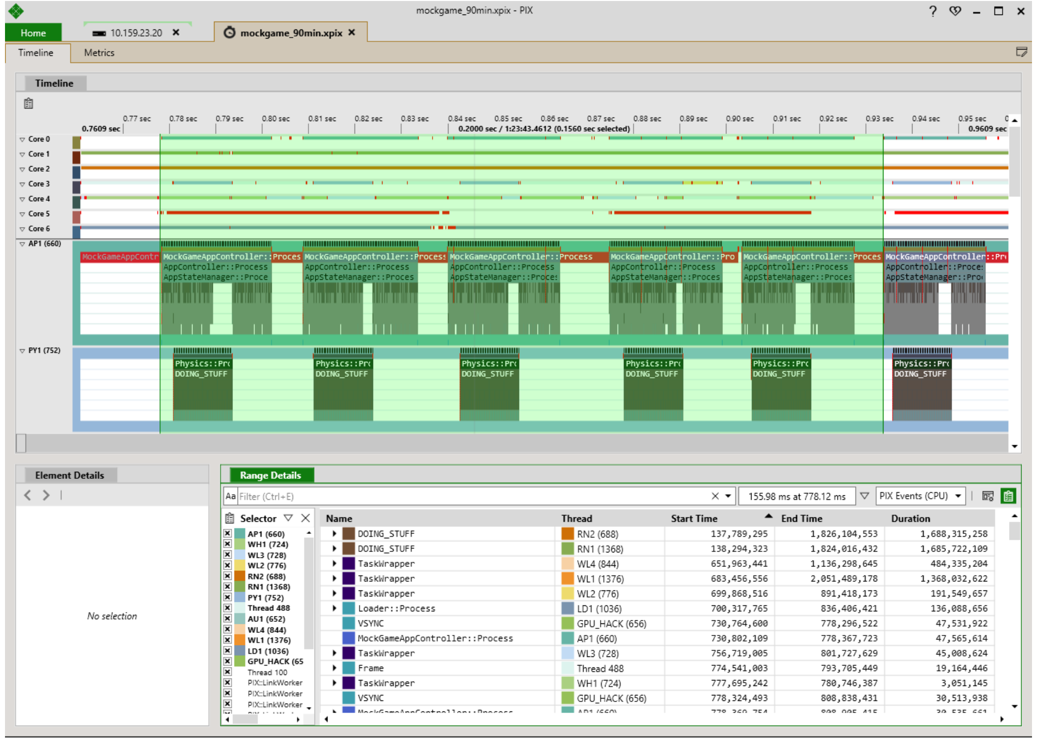


```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

- Deceptive overhead ✕
- « Useful » to know the number of function calls
- Gprof, tyoma/micro-profiler, …
- Read this StackOverflow

Manual instrumentation

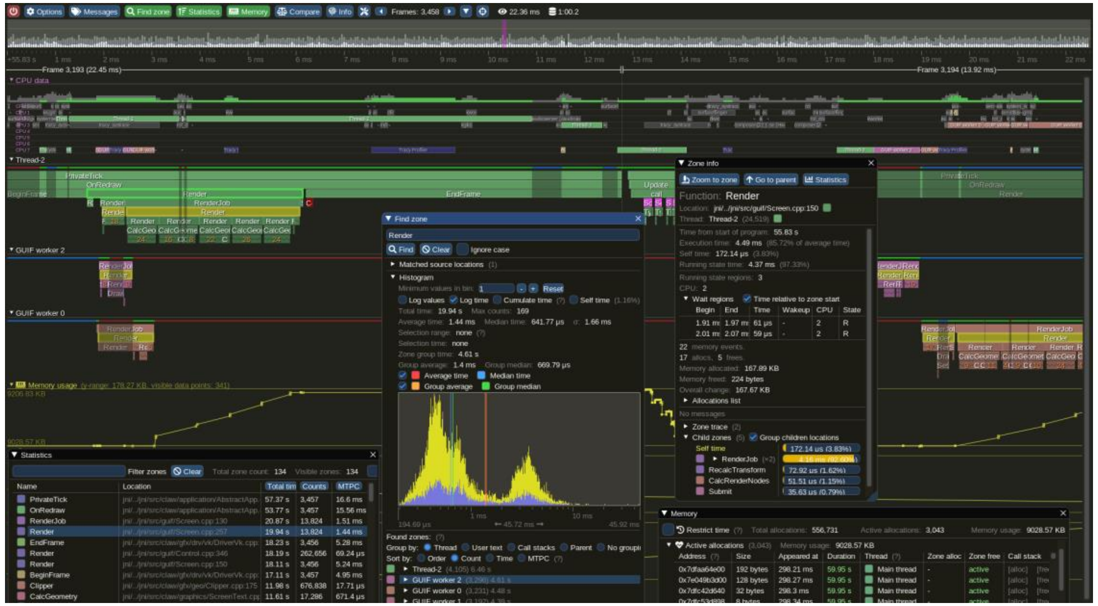


Name	Thread	Start Time	End Time	Duration
DOING_STUFF	RN2 (688)	137,789,295	1,826,104,553	1,688,315,258
DOING_STUFF	RN1 (1368)	138,294,323	1,824,016,432	1,685,722,109
TaskWrapper	WL4 (844)	651,963,441	1,136,298,645	484,335,204
TaskWrapper	WL1 (1376)	683,456,556	2,051,489,178	1,368,032,622
TaskWrapper	WL2 (776)	699,868,516	891,418,173	191,549,657
Loader::Process	LD1 (1036)	700,317,765	836,406,421	136,088,656
VSYNC	GPU_HACK (656)	730,764,600	778,296,522	47,531,922
MockGameAppController::Process	AP1 (660)	730,802,109	778,367,723	47,565,614
TaskWrapper	WL3 (728)	756,719,005	801,727,629	45,008,624
Frame	Thread 488	774,541,003	793,705,449	19,164,446
TaskWrapper	WH1 (724)	777,695,242	780,746,387	3,051,145
VSYNC	GPU_HACK (656)	778,324,493	808,838,431	30,513,938
MockGameAppController::Process	AP1 (660)	778,369,754	808,905,415	30,535,661

- Precise! ✓
- Macro view! ✓
- Business logic hierarchy ✓
- No details ✗
- Requires code changes ✗
- Remotery, easy_profiler, …

Instrumentation + Sampling

Best of both worlds



- Microsoft PIX for Windows
- Tracy
- Optick
- Superluminal
- Perfetto
- …

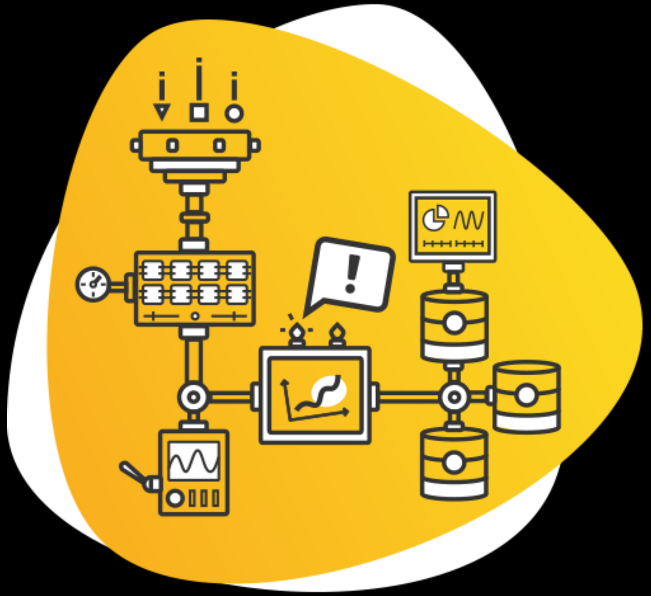
Profiler wishlist

- ❑ Lightweight
- ❑ Accurate
- ❑ Multi-threading support
  - ❑ Thread naming
  - ❑ Context switch details
- ❑ I/O measurements
- ❑ Memory measurements
- ❑ Custom data sources
- ❑ Cross-platform
- ❑ A good UI/UX
- ❑ Can save sessions
- ❑ Fibers/coroutines support?
- ❑ Can be attached to a running app
- ❑ CI support?
- ❑ …

siliceum

Macro-optimization

Looking at the big picture



Compilation flags

Quick to check and test

GCC/Clang

- -O3
- -march=native
- -flto
- -ffast-math ⚠

MSVC

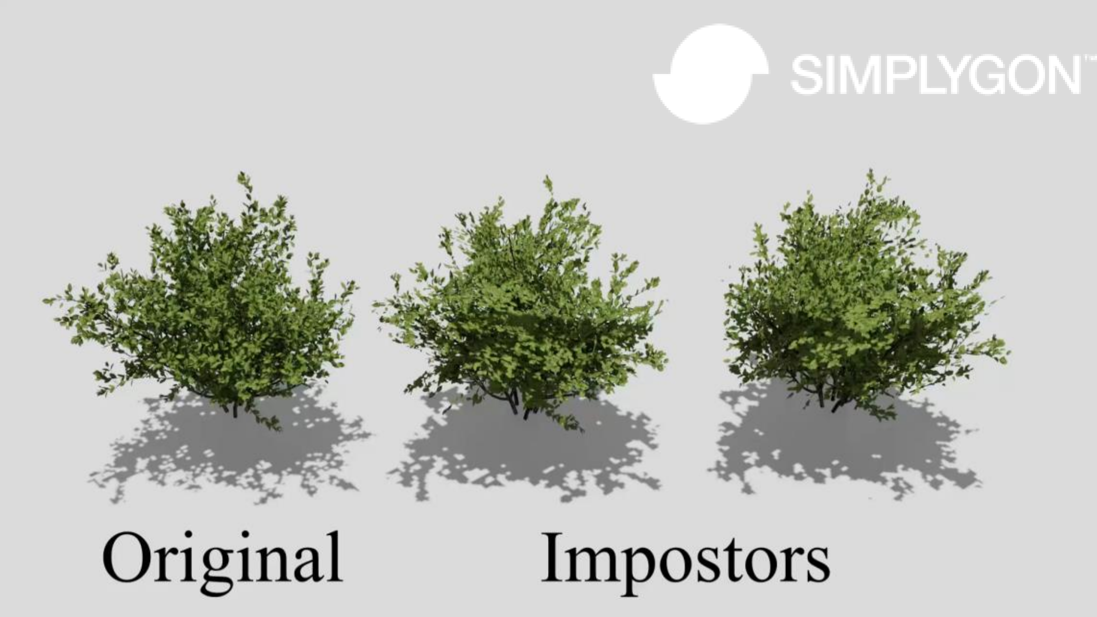
- /O2
- /arch:???
- /GL, /LTCG
- /fp:fast ⚠

Do nothing

The best of optimizations

- Avoid unneeded processing
- Avoid unneeded initialization/copies
- Avoid unneeded (re)allocations (use `reserve()`!)
- Avoid using mutexes (and reduce the code it guards)
- Avoid duplicated function calls (abstractions!)

Adapt your data or design



Adapt your data or design

Yann Richard ©

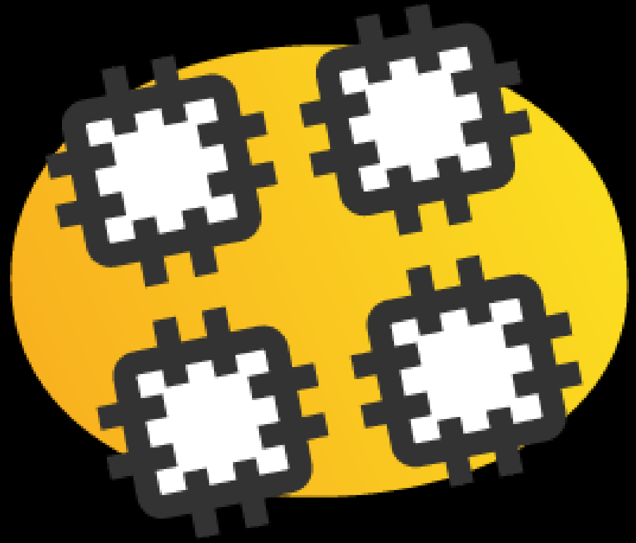
In order of efficiency:

- Do less
- Do it faster
- Prepare data (compress, pre-compute, …)
- Multi-core (parallelism, latency hiding, …)
- Caching (do this last or you'll regret it!)

siliceum

War story:
WebKit GC in a game

WebKit and real-time



Some context

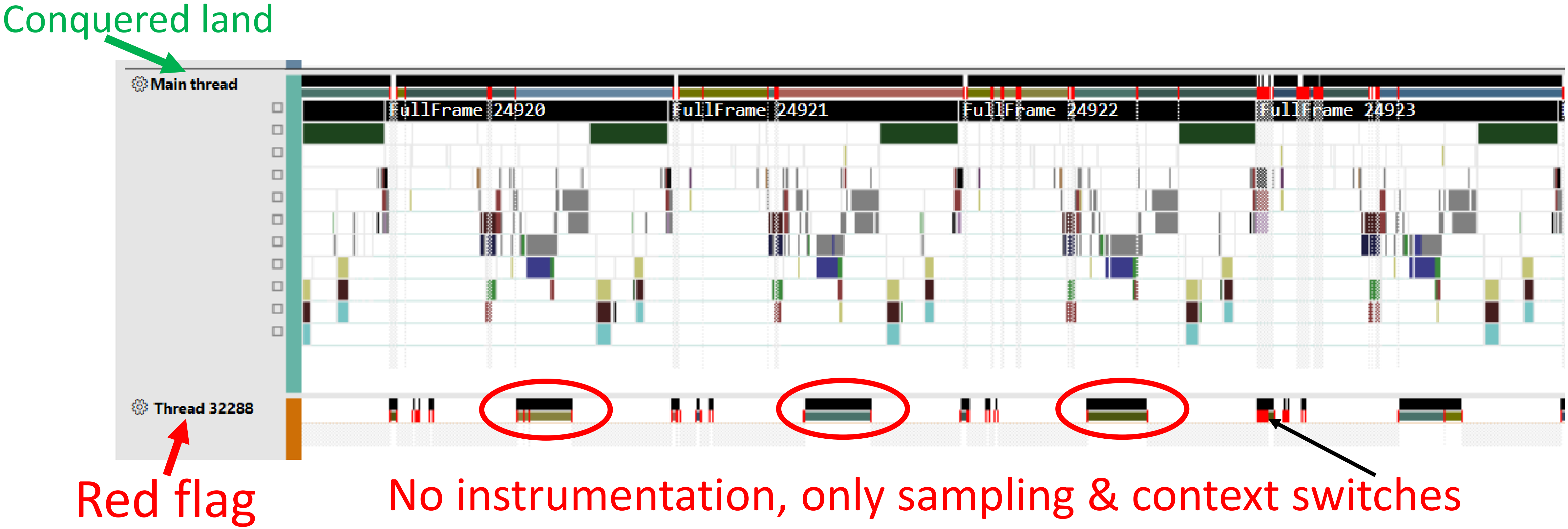
- WebKit used to display dynamic UIs, game components…
- Target for a game is 60|30fps => 33|16ms
- No JIT on console
- « Old » WebKit version from ~2016
- Frequent freezes could be observed

Remember:

Measure 1st

(we already have our target: 60/30fps)

Start collecting data

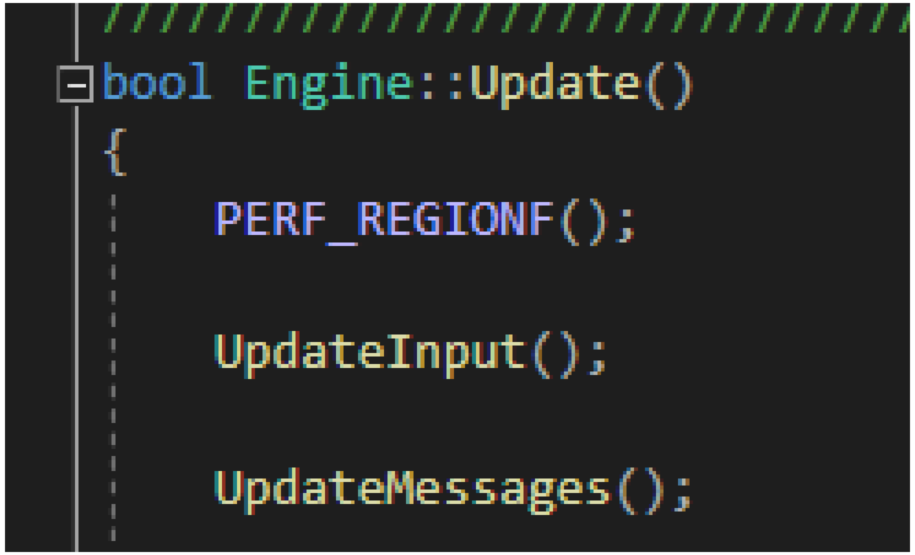


Conquered land

Red flag

No instrumentation, only sampling & context switches

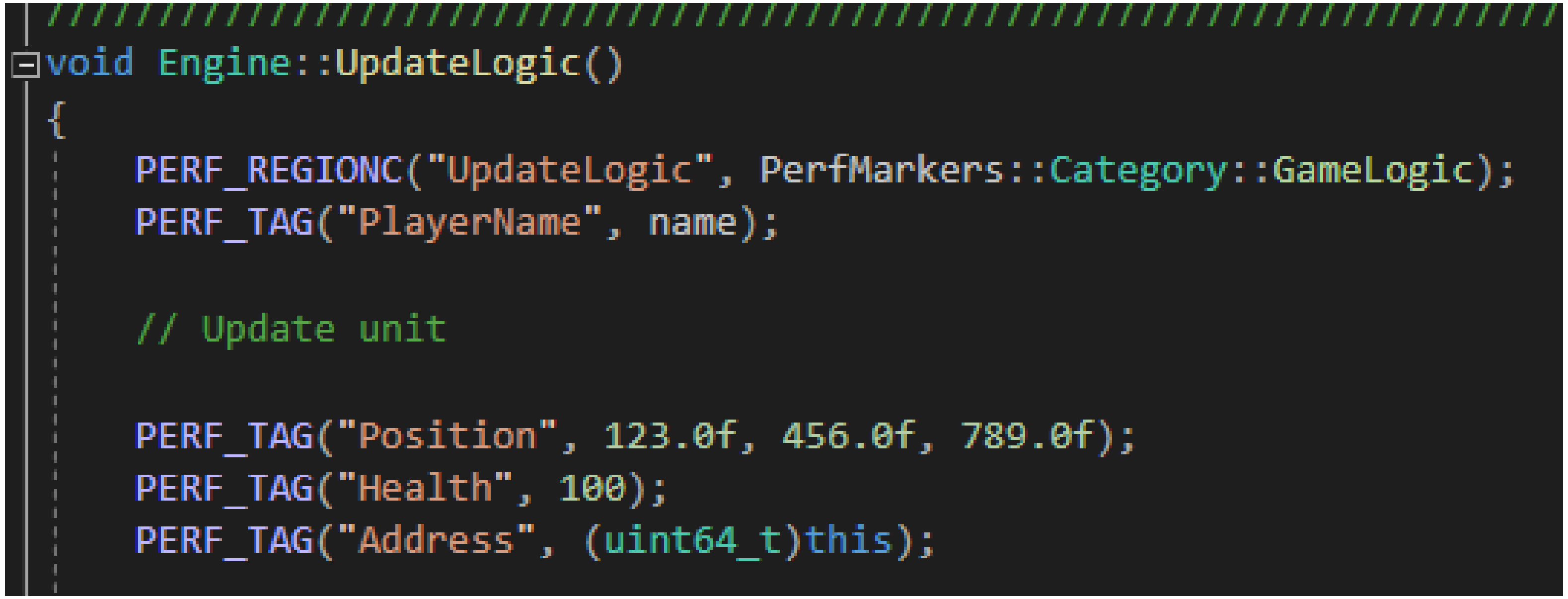
What it looks like in the code



```
////////////////////////////////////
bool Engine::Update()
{
    PERF_REGIONF();

    UpdateInput();

    UpdateMessages();
```

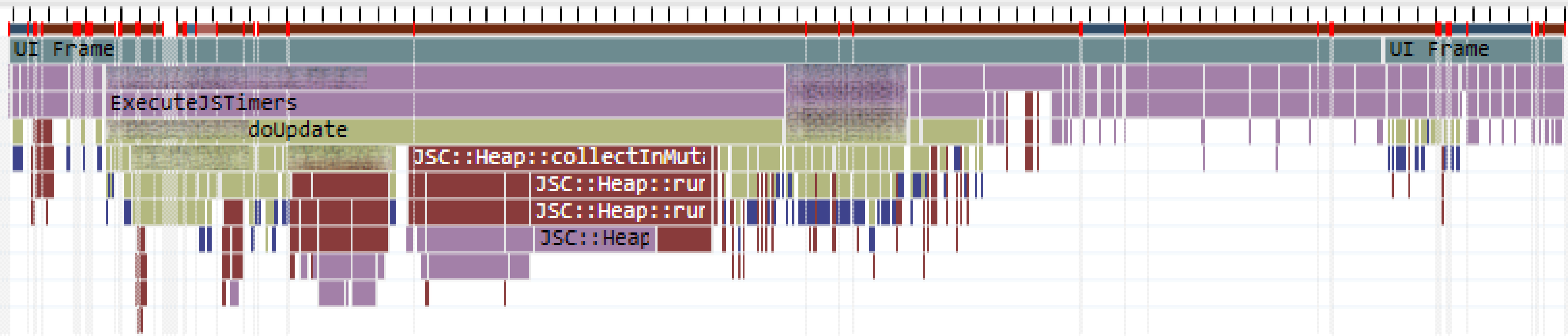


```
////////////////////////////////////////////////////////////
void Engine::UpdateLogic()
{
    PERF_REGIONC("UpdateLogic", PerfMarkers::Category::GameLogic);
    PERF_TAG("PlayerName", name);

    // Update unit

    PERF_TAG("Position", 123.0f, 456.0f, 789.0f);
    PERF_TAG("Health", 100);
    PERF_TAG("Address", (uint64_t)this);
```

The garbage collector monster



What do we do now?

Many options

❑Tweak the garbage collector parameters
❑Optimize the garbage collector
❑Make it less blocking
❑Rewrite the code in C++
❑Reduce the amount of data to process
~~❑Ship it~~

Reduce the amount of data to process

Big wins without diving into WebKit

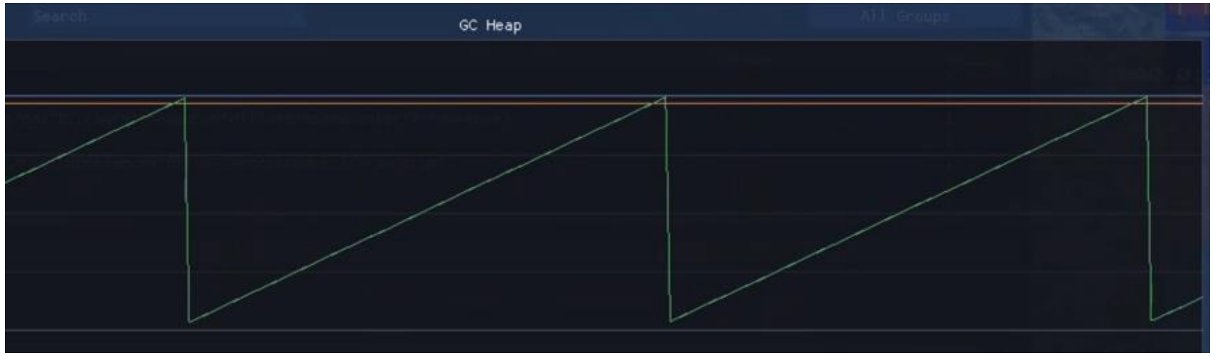
- Analyze what generates memory usage
- Fix JS code to generate less memory allocations
- Sadly not enough
  - Still no control over when the GC will happen
  - Still have frequent « freezes » due to GC

Eden vs Full GC

- Sometimes GC takes 70ms, sometimes « only » 10ms?
- Full GC
    - Iterates over all allocations!
- Eden GC
    - Iterates only over objects allocated/changed since the previous GC
    - This is what we want most of the time in a game loop

Avoiding full GC

- Full GC always triggered after eden
- Solution:
  Trigger only if size after GC is
  - above full GC threshold
  - X% greater than previous post-full GC size

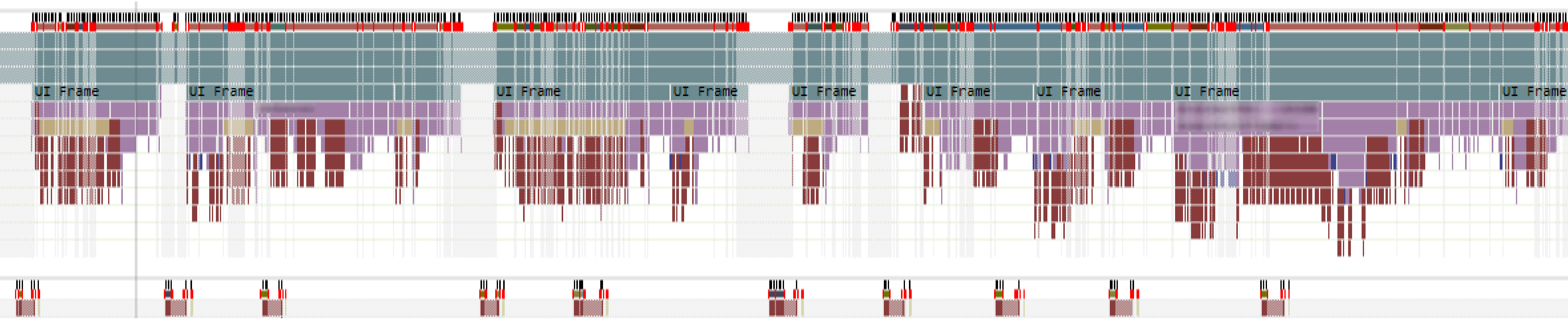


🟧 Eden GC size threshold

🟩 Current memory size

GC may pause execution to run

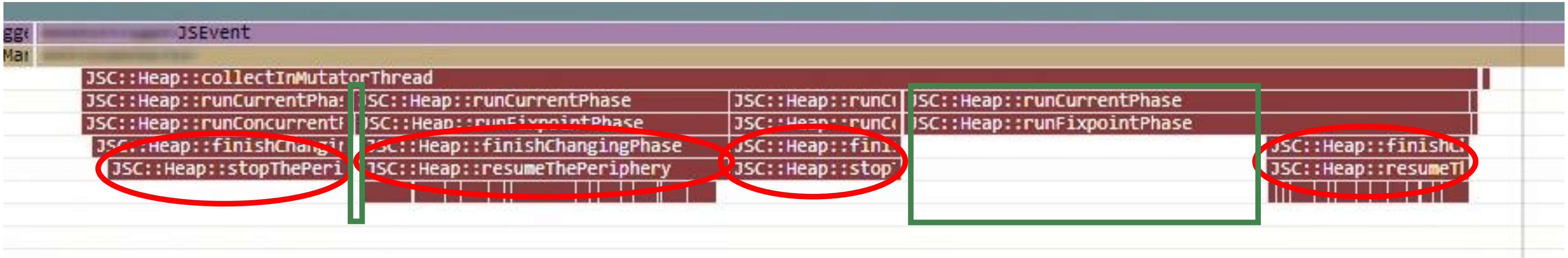
Scheduler not a good fit for realtime purposes



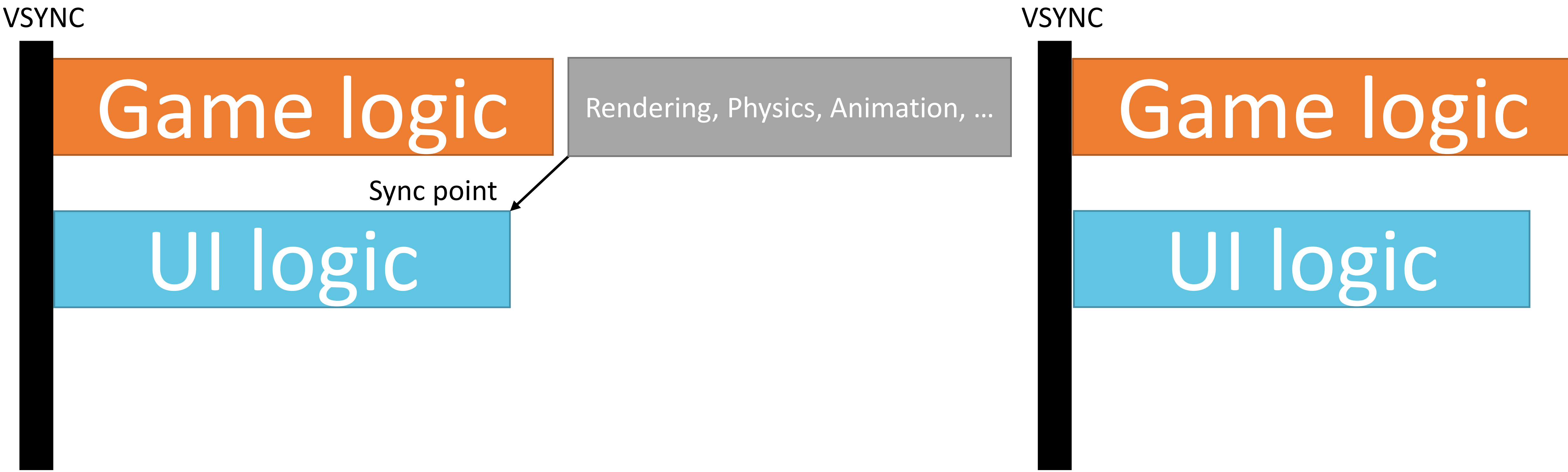
Prologue & epilogue of GC passes are expensive

○ stop/resumeThePeriphery

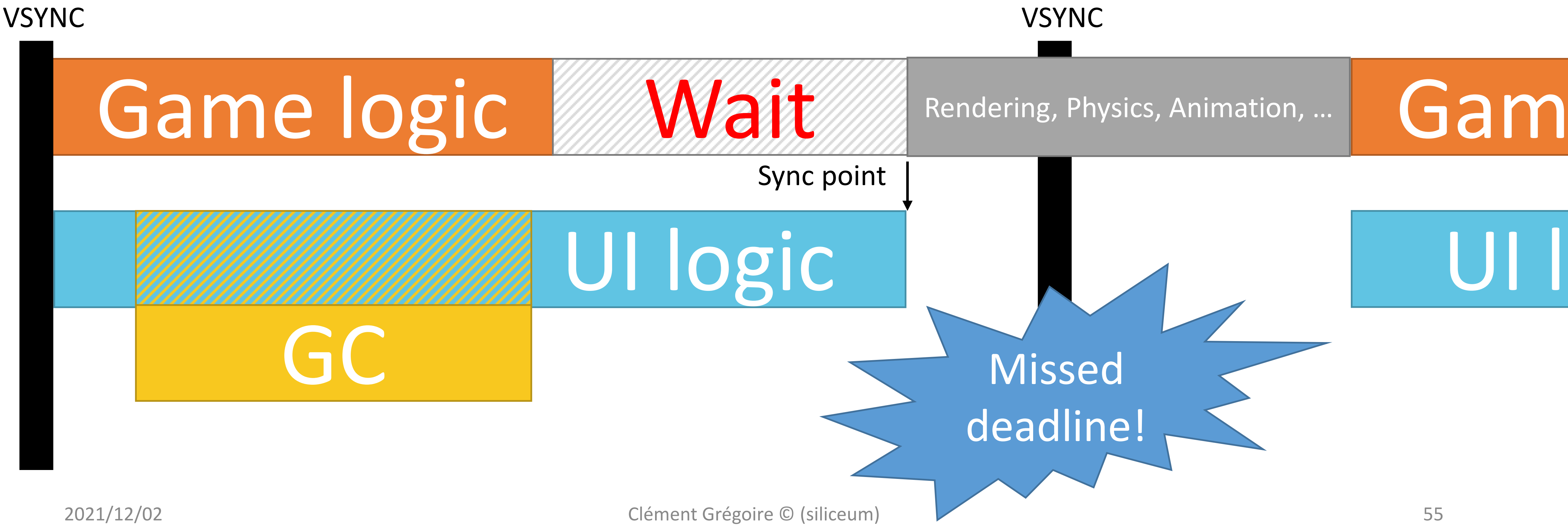
🟩 Actual useful work


JSEvent
JSC::Heap::collectInMutatorThread
JSC::Heap::runCurrentPhas
JSC::Heap::runConcurrentF
JSC::Heap::finishChangi
JSC::Heap::stopThePeri
JSC::Heap::runCurrentPhase
JSC::Heap::runFixpointPhase
JSC::Heap::finishChangingPhase
JSC::Heap::resumeThePeriphery
JSC::Heap::runC
JSC::Heap::runC
JSC::Heap::fini
JSC::Heap::stop
JSC::Heap::runCurrentPhase
JSC::Heap::runFixpointPhase
JSC::Heap::finishC
JSC::Heap::resumeT

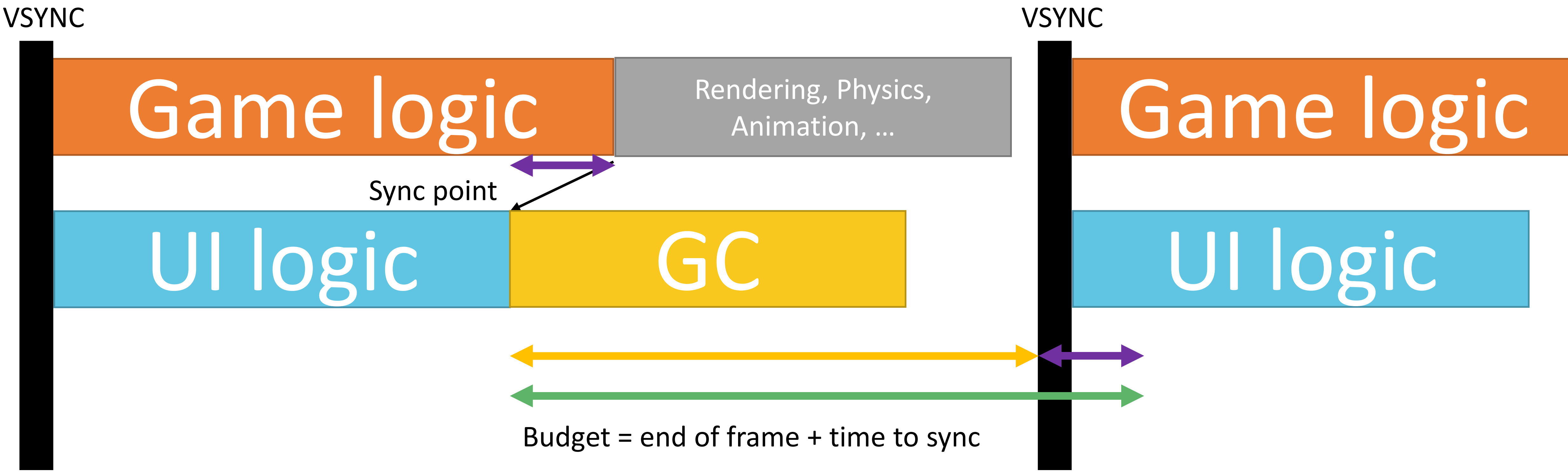
A normal frame



Here be dragons (GC)

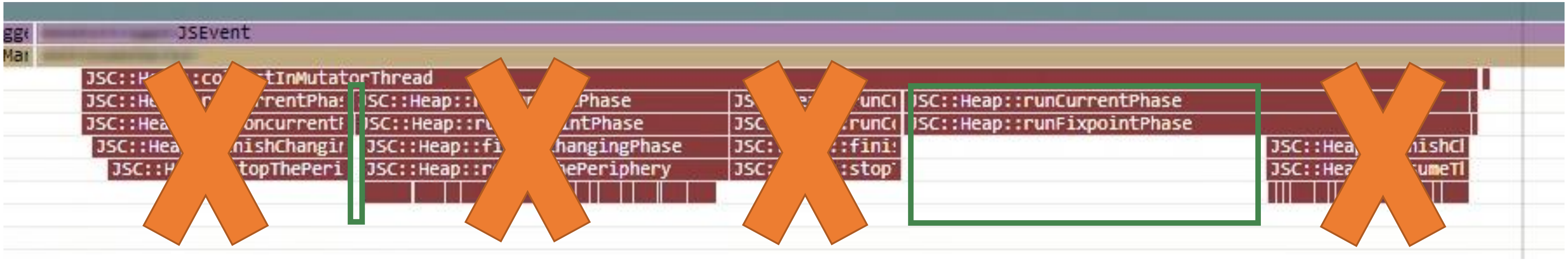


The solution



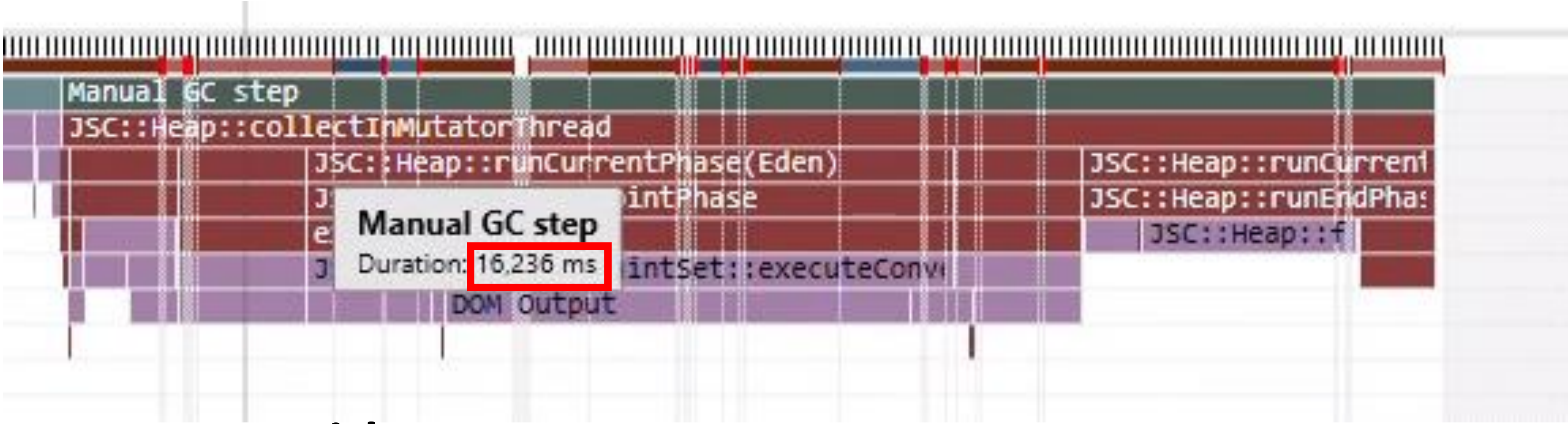
Budget = end of frame + time to sync

Bonus point


JSEvent
JSC::H :co tInMutatorThread
JSC::He rrentPhas
JSC::Hea oncurrentP
JSC::Hea nishChangin
JSC::H topThePeri
JSC::Heap:: Phase
JSC::Heap::r intPhase
JSC::Heap::fi hangingPhase
JSC::Heap::r ePeriphery
JS unC
JSC :runC
JSC: ::fini
JSC: :stop
JSC::Heap::runCurrentPhase
JSC::Heap::runFixpointPhase
JSC::Hea ishCl
JSC::Hea umeTl

Update JSCore to benefit from multi-threading

Take advantage of all the cores!

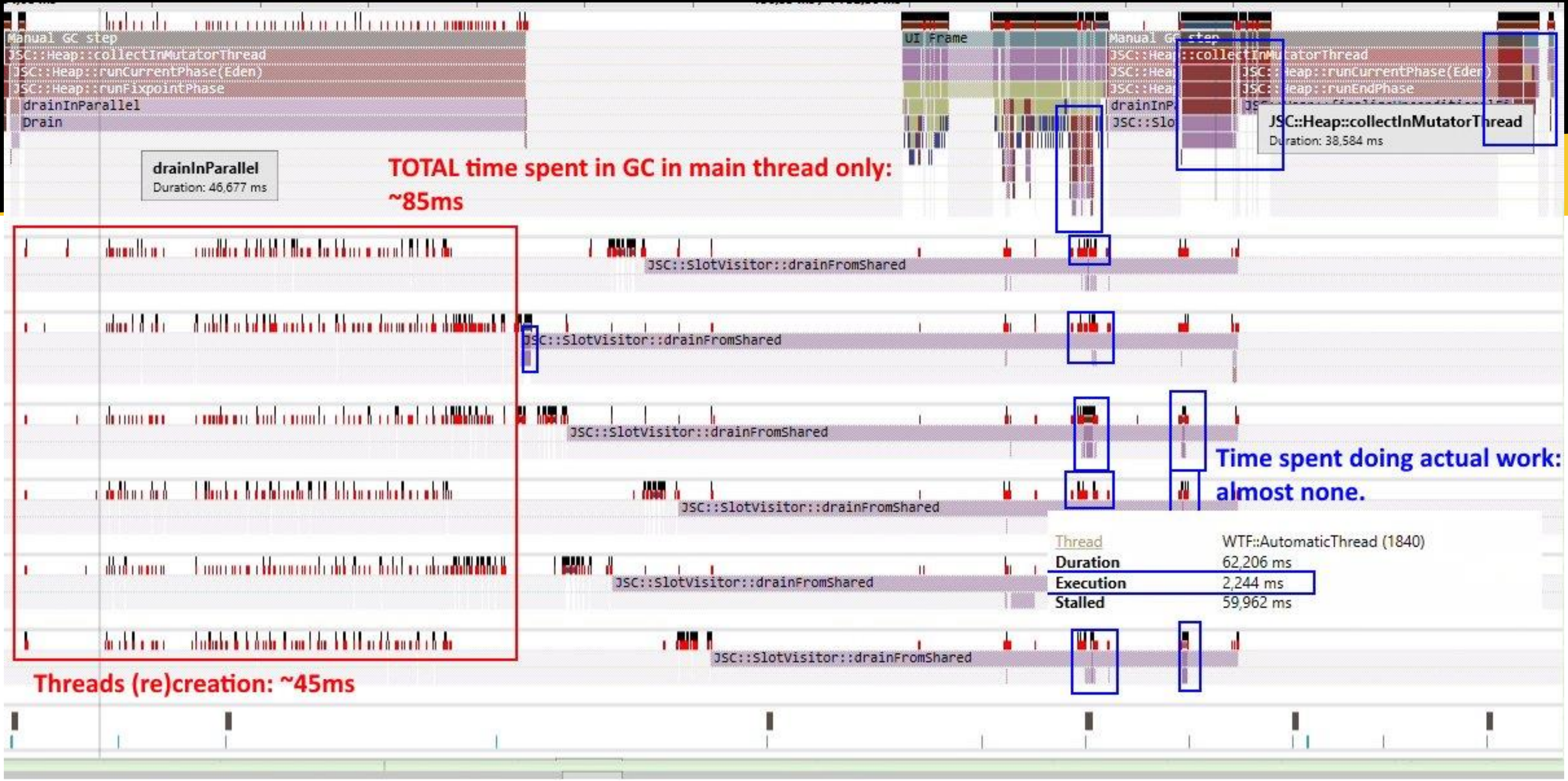


Manual GC step

Duration: 16,236 ms

GC step **without** MT

Update JSCore to benefit from MT?


Manual GC step
JSC::Heap::collectInMutatorThread
JSC::Heap::runCurrentPhase(Eden)
JSC::Heap::runFixpointPhase
drainInParallel
Drain
UI Frame
JSC::Heap::runEndPhase
JSC::SlotVisitor::drainFromShared

drainInParallel
Duration: 46,677 ms

JSC::Heap::collectInMutatorThread
Duration: 38,584 ms

TOTAL time spent in GC in main thread only: ~85ms

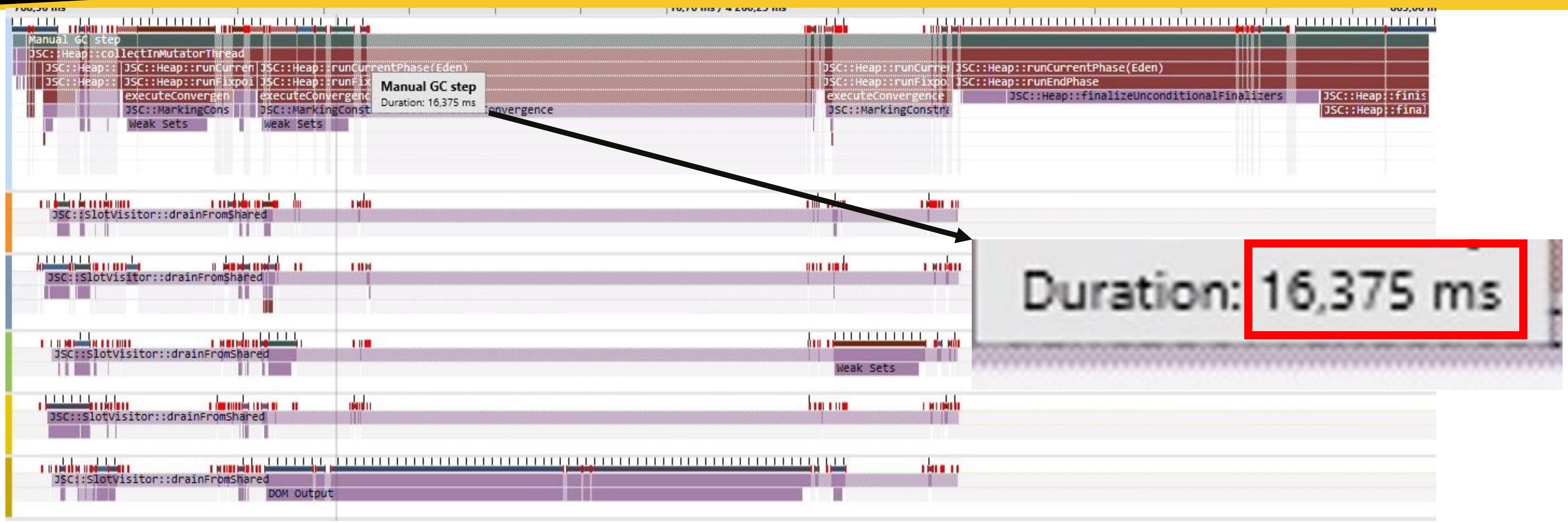
Time spent doing actual work: almost none.

Thread	WTF::AutomaticThread (1840)
Duration	62,206 ms
Execution	2,244 ms
Stalled	59,962 ms

Threads (re)creation: ~45ms

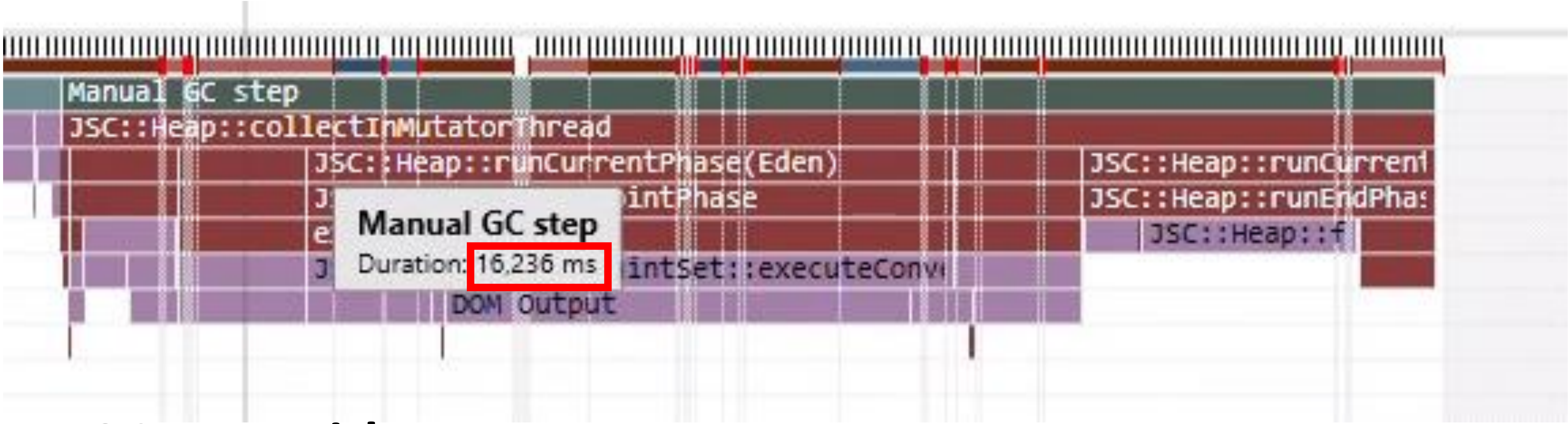
Update JavaScriptCore to benefit from MT

Does not scale as-is


Manual GC step
JSC::Heap::collectInMutatorThread
JSC::Heap::runCurrentPhase(Eden)
JSC::Heap::runFixpoi
executeConvergen
JSC::MarkingCons
Weak Sets
Manual GC step
Duration: 16,375 ms
onvergence
JSC::Heap::runCurrentPhase(Eden)
JSC::Heap::runEndPhase
JSC::Heap::finalizeUnconditionalFinalizers
JSC::Heap::finis
JSC::Heap::final
JSC::SlotVisitor::drainFromShared
JSC::SlotVisitor::drainFromShared
JSC::SlotVisitor::drainFromShared
JSC::SlotVisitor::drainFromShared
JSC::SlotVisitor::drainFromShared
DOM Output
Weak Sets
Duration: 16,375 ms

~~Update JSCore to benefit from multi-threading~~

The wild goose chase



GC step **without** MT

Many more optimizations

Not really C++

- Reduce string copies for C++ ⇔ .js communications
- Reduced allocations and indirections in WebKit
- No getters/setters => slow path in VM
- Reduce DOM interactions
- Functions forcing layout (Element.getBoundingClientRect())

If you had to remember one thing

Measure, measure, measure!

(and understand what you measure)

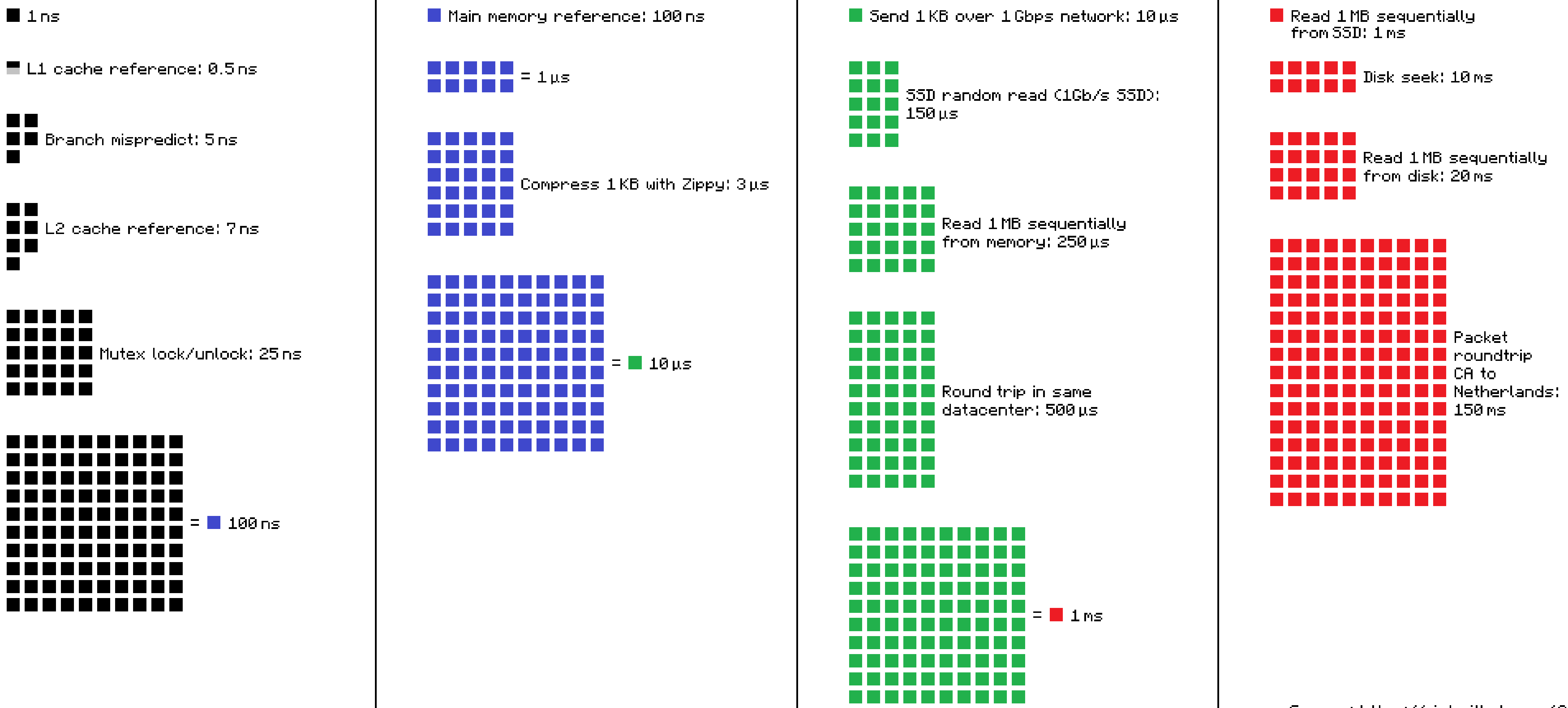
Thank you!
(contact me)

Email: clement@siliceum.com

Twitter: @lectem
Discord: #include

Appendix

Latency Numbers Every Programmer Should Know



⬛ 1 ns

L1 cache reference: 0.5 ns

Branch mispredict: 5 ns

L2 cache reference: 7 ns

Mutex lock/unlock: 25 ns

= 🟦 100 ns

🟦 Main memory reference: 100 ns

= 1 µs

Compress 1 KB with Zippy: 3 µs

= 🟩 10 µs

🟩 Send 1 KB over 1 Gbps network: 10 µs

SSD random read (1Gb/s SSD): 150 µs

Read 1 MB sequentially from memory: 250 µs

Round trip in same datacenter: 500 µs

= 🟥 1 ms

🟥 Read 1 MB sequentially from SSD: 1 ms

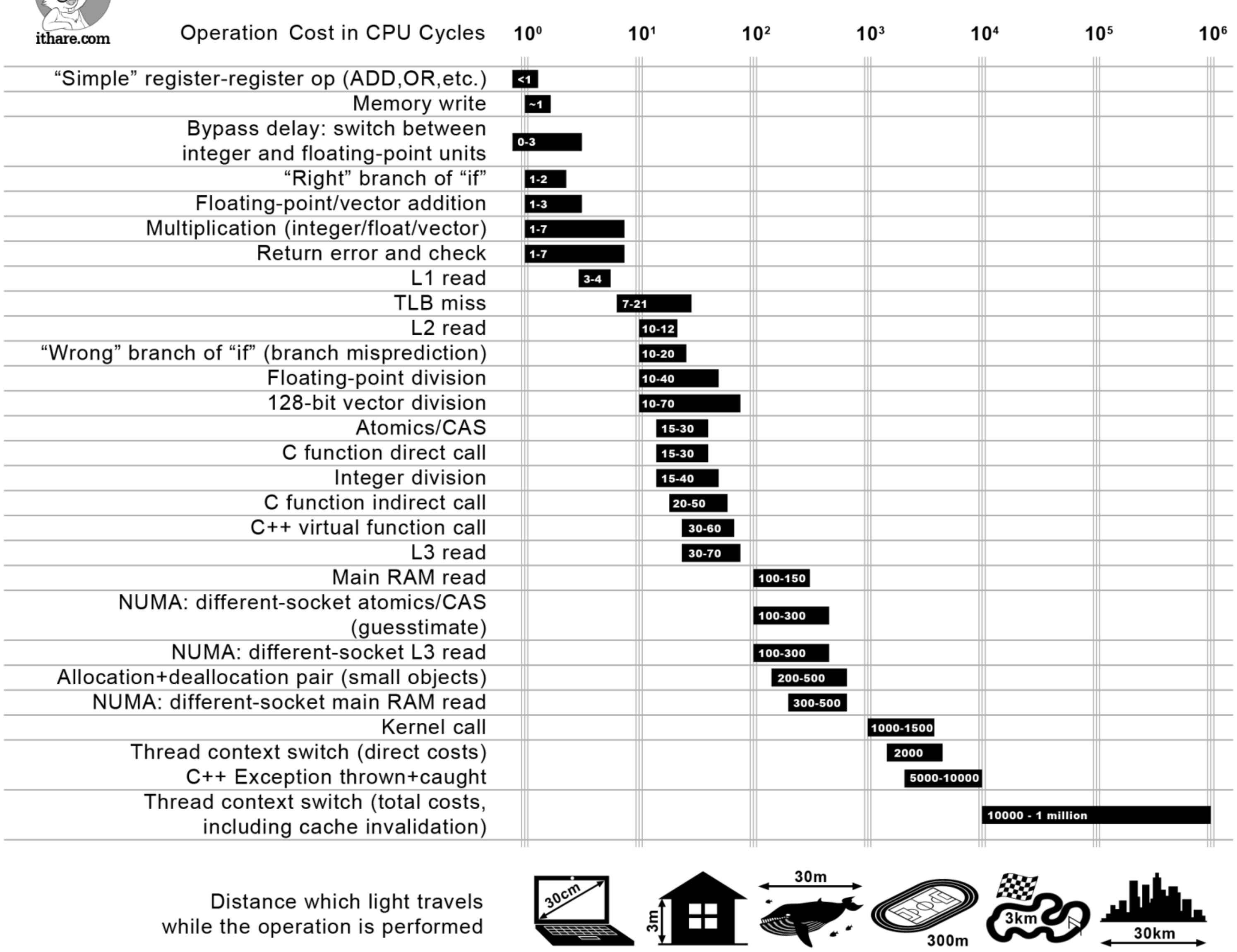
Disk seek: 10 ms

Read 1 MB sequentially from disk: 20 ms

Packet roundtrip CA to Netherlands: 150 ms

Source: https://gist.github.com/2841832

Not all CPU operations are created equal


ithare.com
Operation Cost in CPU Cycles
10^0
10^1
10^2
10^3
10^4
10^5
10^6
"Simple" register-register op (ADD,OR,etc.) <1
Memory write ~1
Bypass delay: switch between integer and floating-point units 0-3
"Right" branch of "if" 1-2
Floating-point/vector addition 1-3
Multiplication (integer/float/vector) 1-7
Return error and check 1-7
L1 read 3-4
TLB miss 7-21
L2 read 10-12
"Wrong" branch of "if" (branch misprediction) 10-20
Floating-point division 10-40
128-bit vector division 10-70
Atomics/CAS 15-30
C function direct call 15-30
Integer division 15-40
C function indirect call 20-50
C++ virtual function call 30-60
L3 read 30-70
Main RAM read 100-150
NUMA: different-socket atomics/CAS (guesstimate) 100-300
NUMA: different-socket L3 read 100-300
Allocation+deallocation pair (small objects) 200-500
NUMA: different-socket main RAM read 300-500
Kernel call 1000-1500
Thread context switch (direct costs) 2000
C++ Exception thrown+caught 5000-10000
Thread context switch (total costs, including cache invalidation) 10000 - 1 million
Distance which light travels while the operation is performed
30cm
3m
30m
300m
3km
30km