

C++ ONLINE

DAVE ROWLAND

LESSONS LEARNED
FROM OPTIMISING A
HIGH PERFORMANCE
LIBRARY

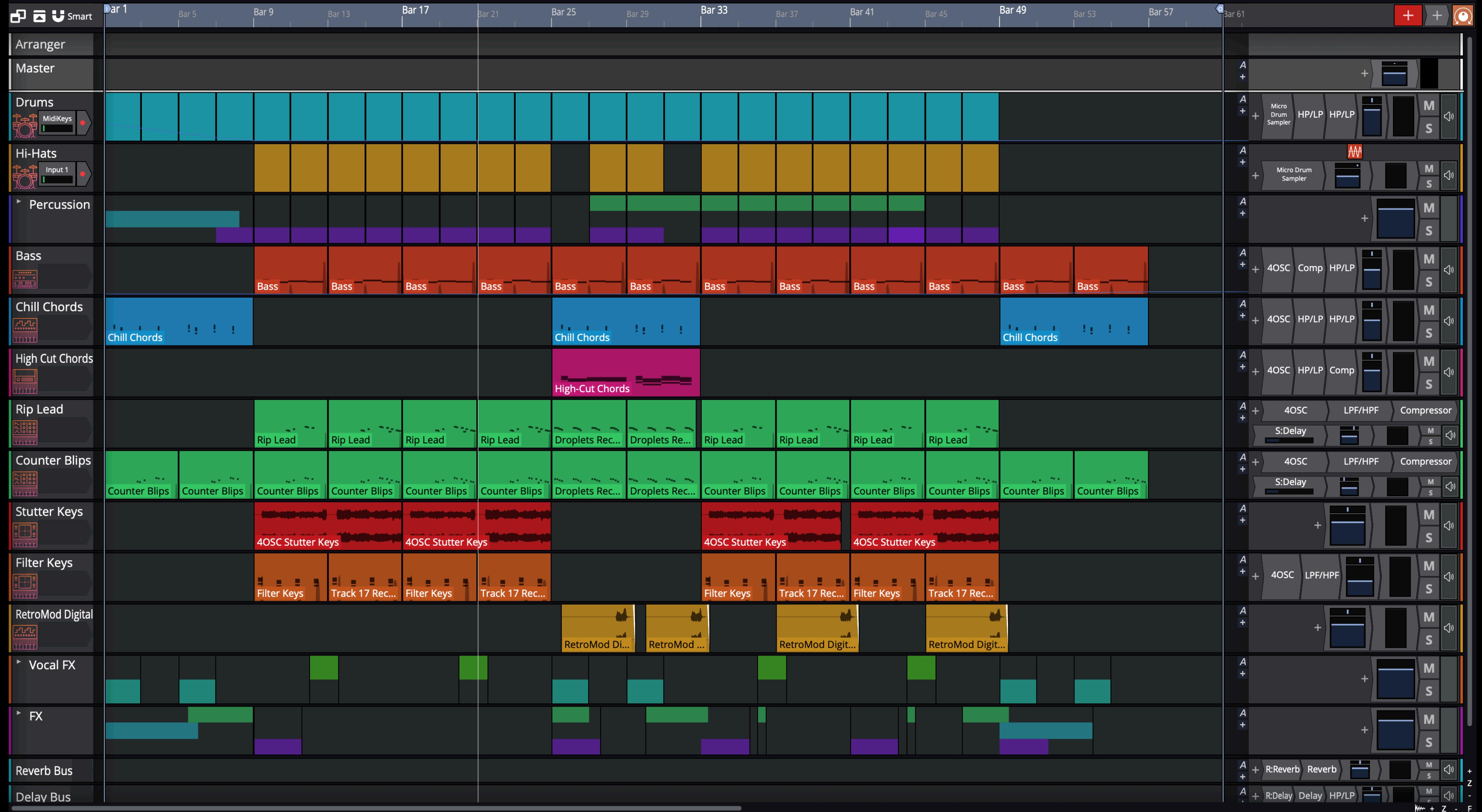
2024

Lessons Learned from Optimising a High Performance Library

Dave Rowland

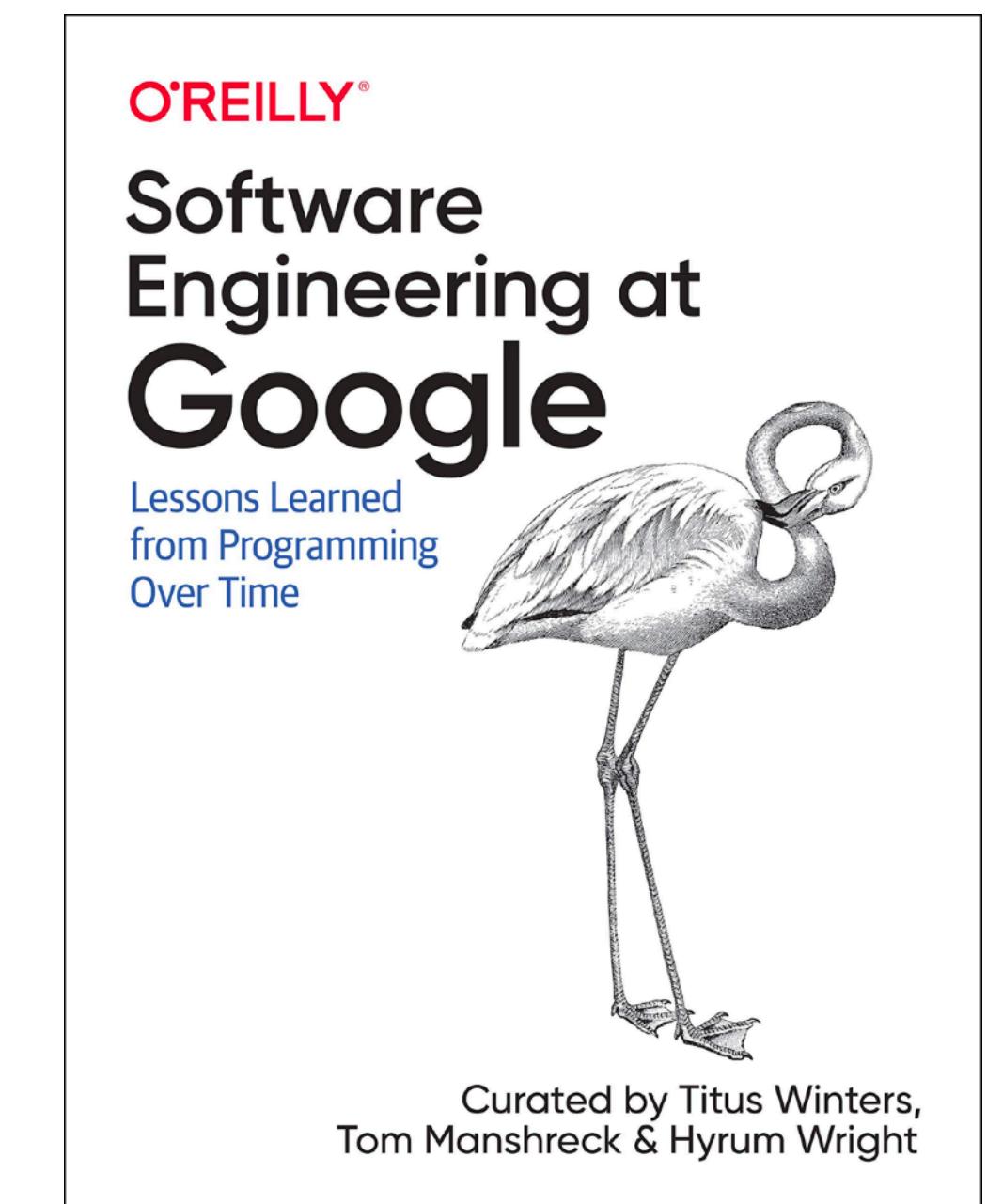
X @drowaudio, 🎵 drowaudio





“Software engineering is programming integrated over time”

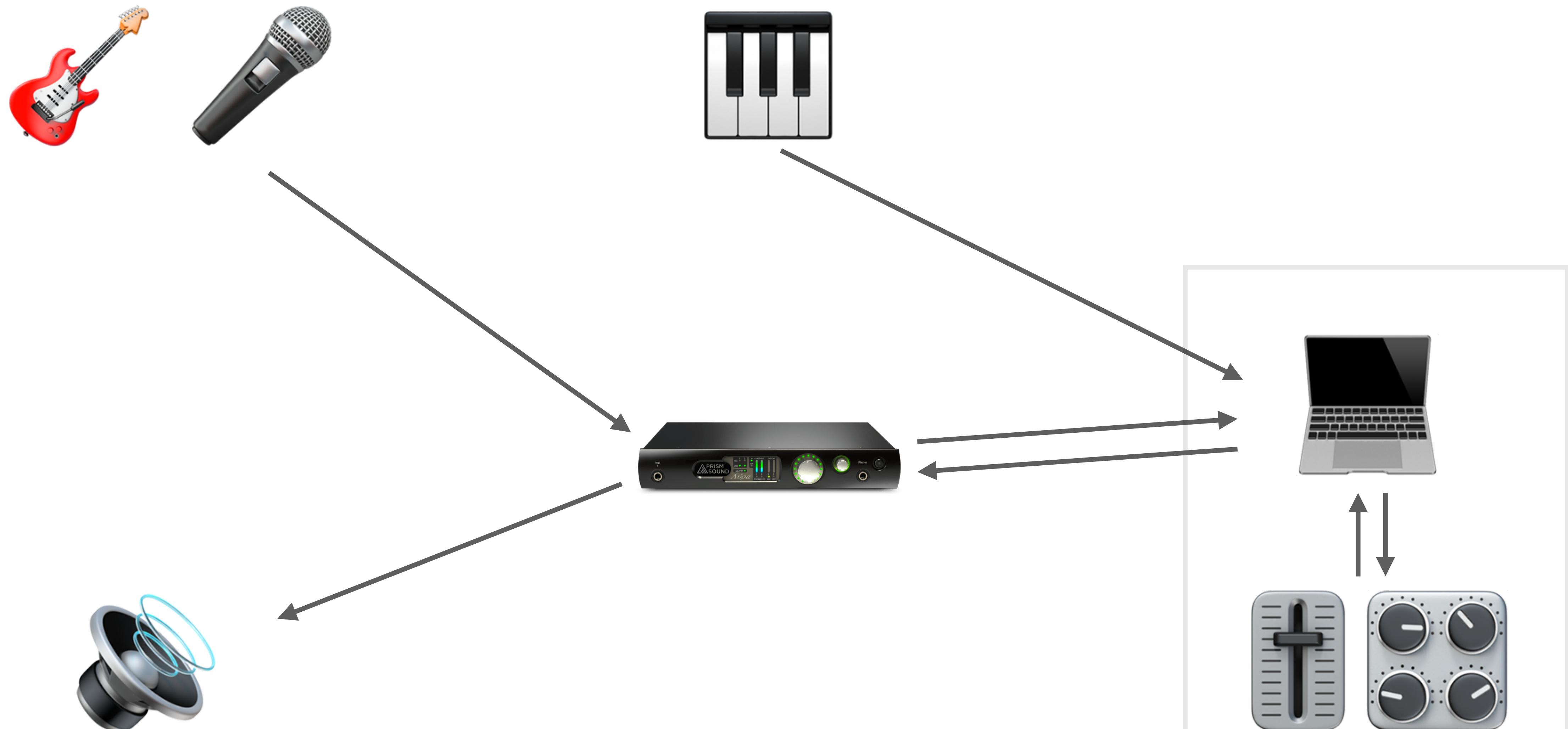
Titus Winters

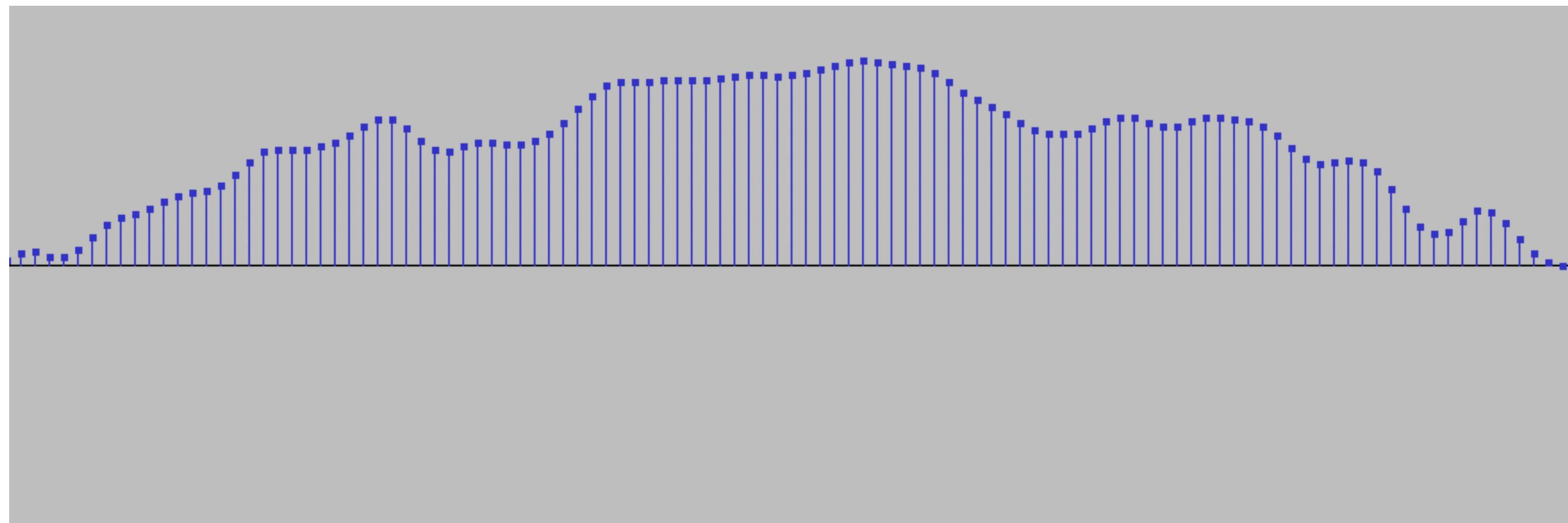
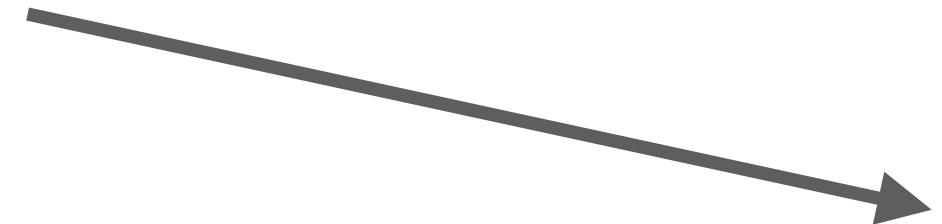
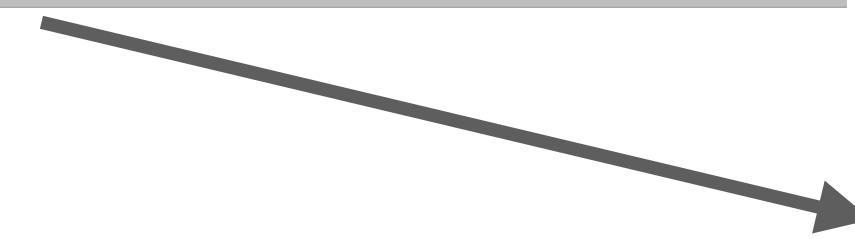
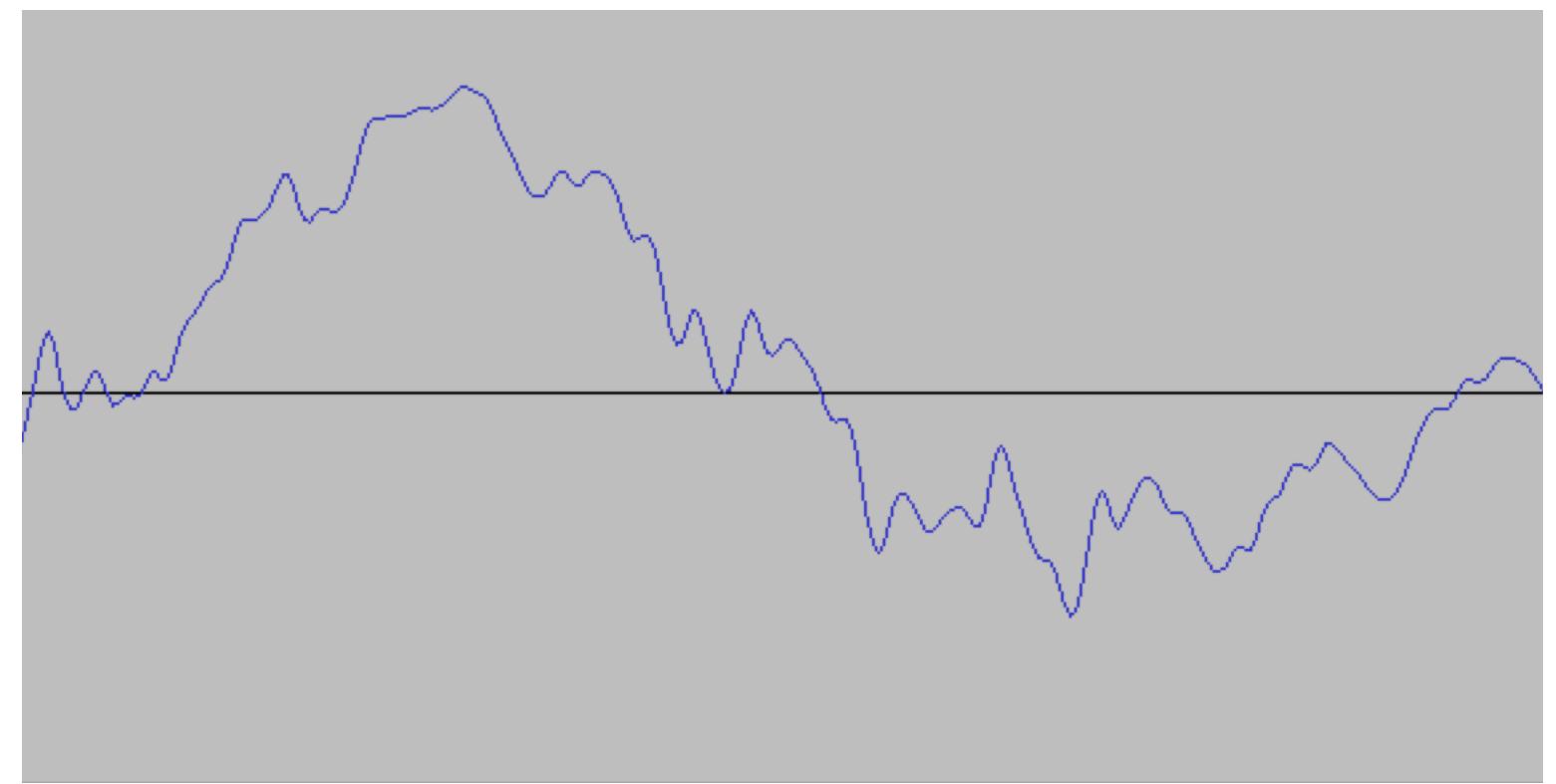


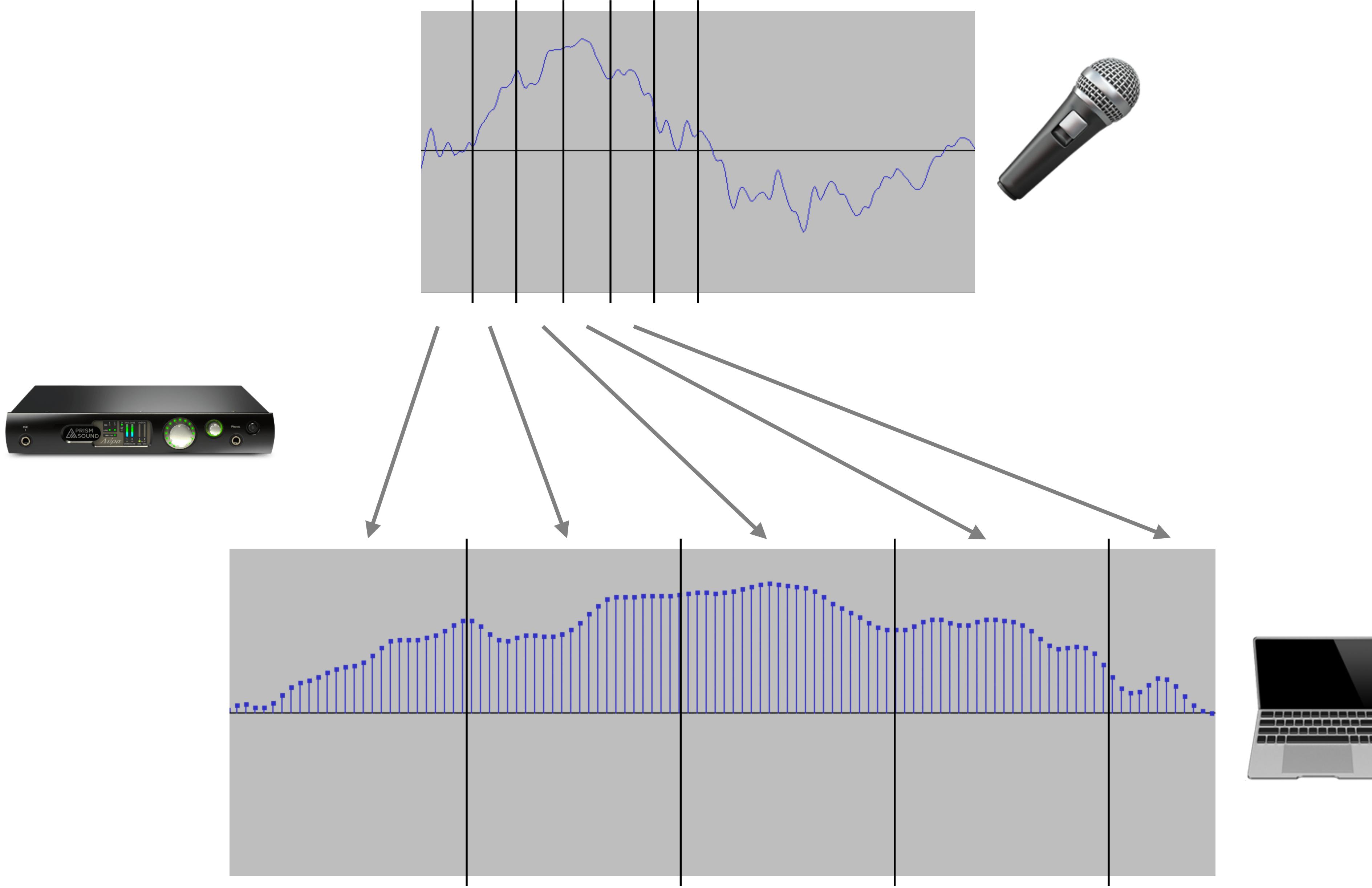
Chapters

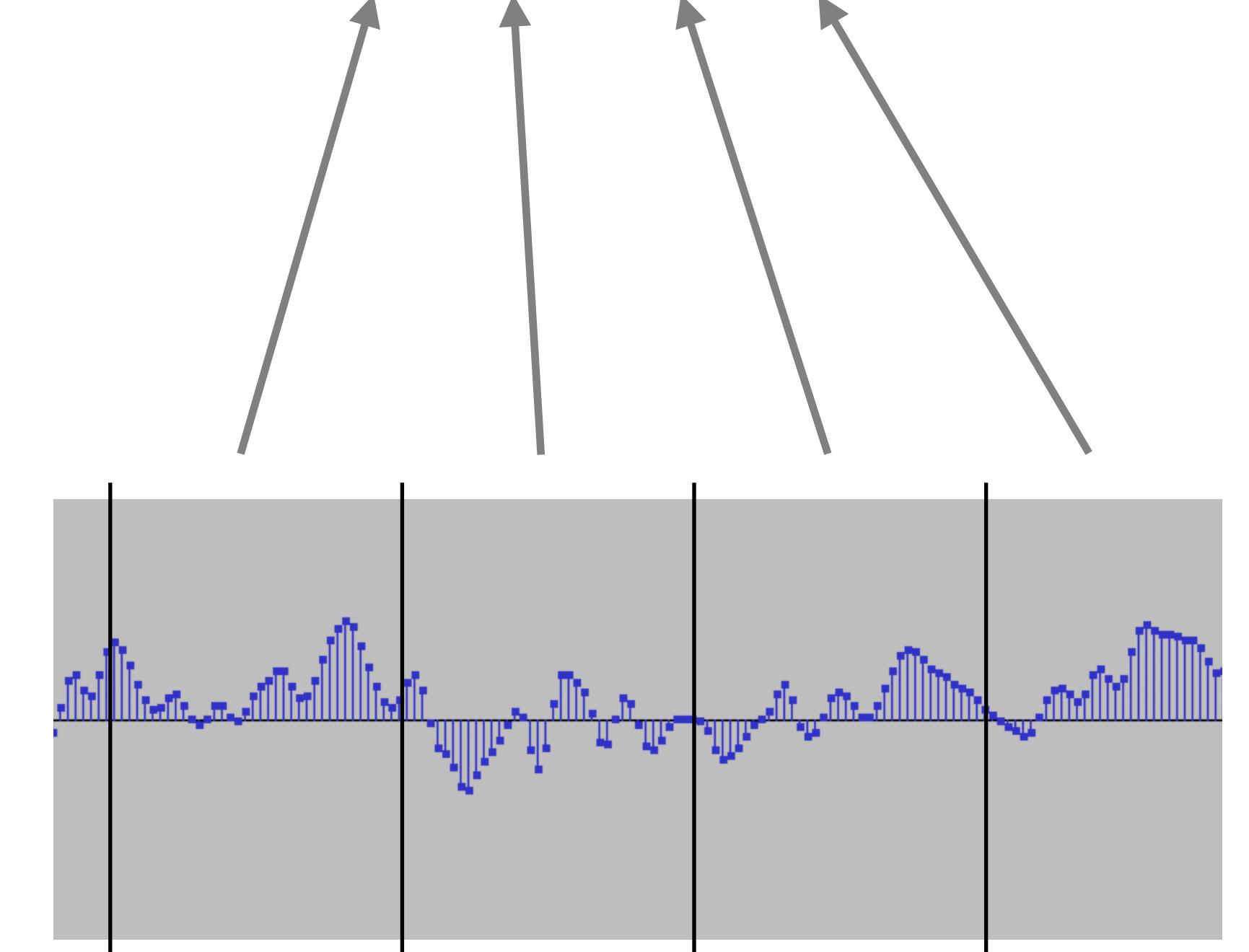
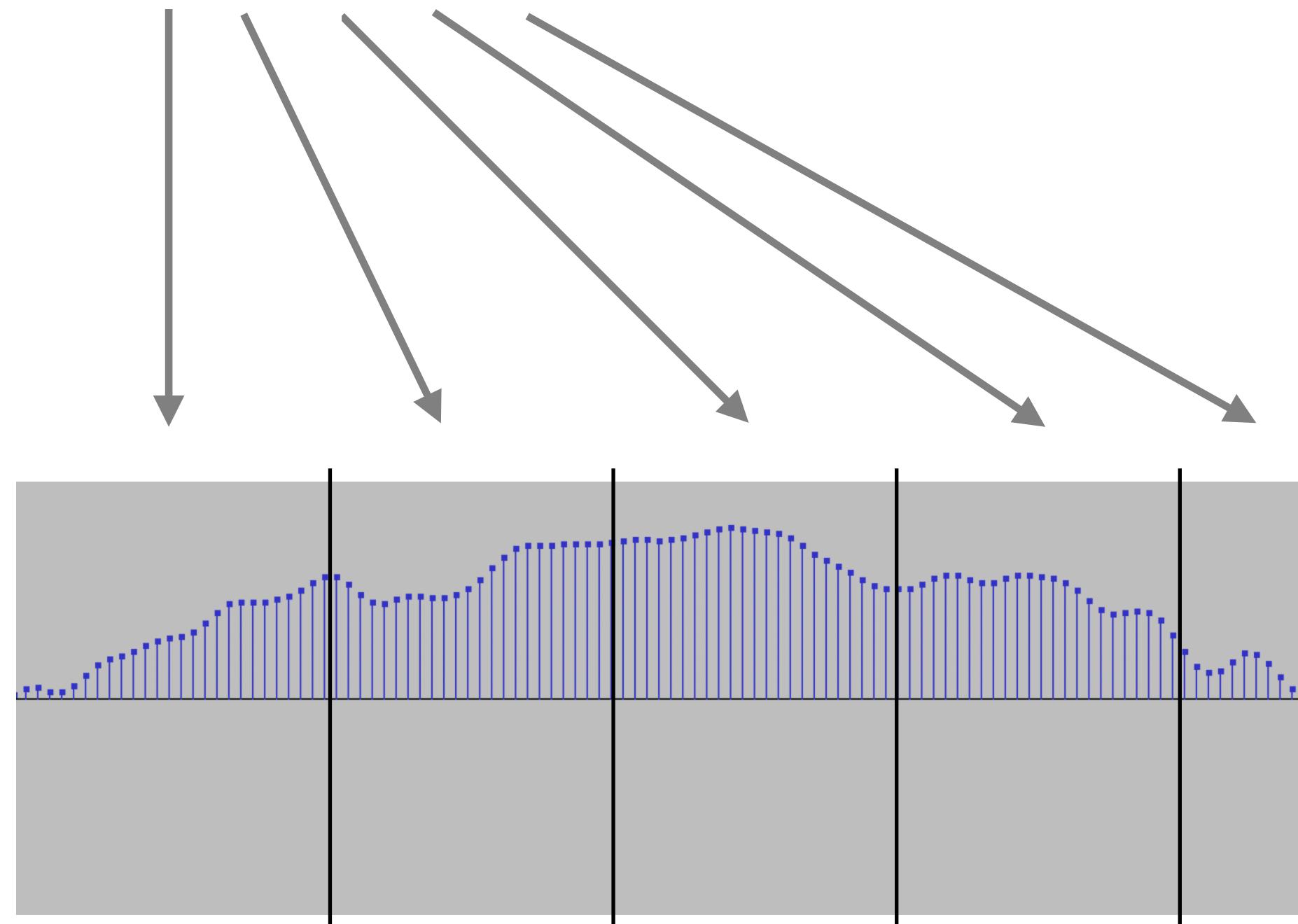
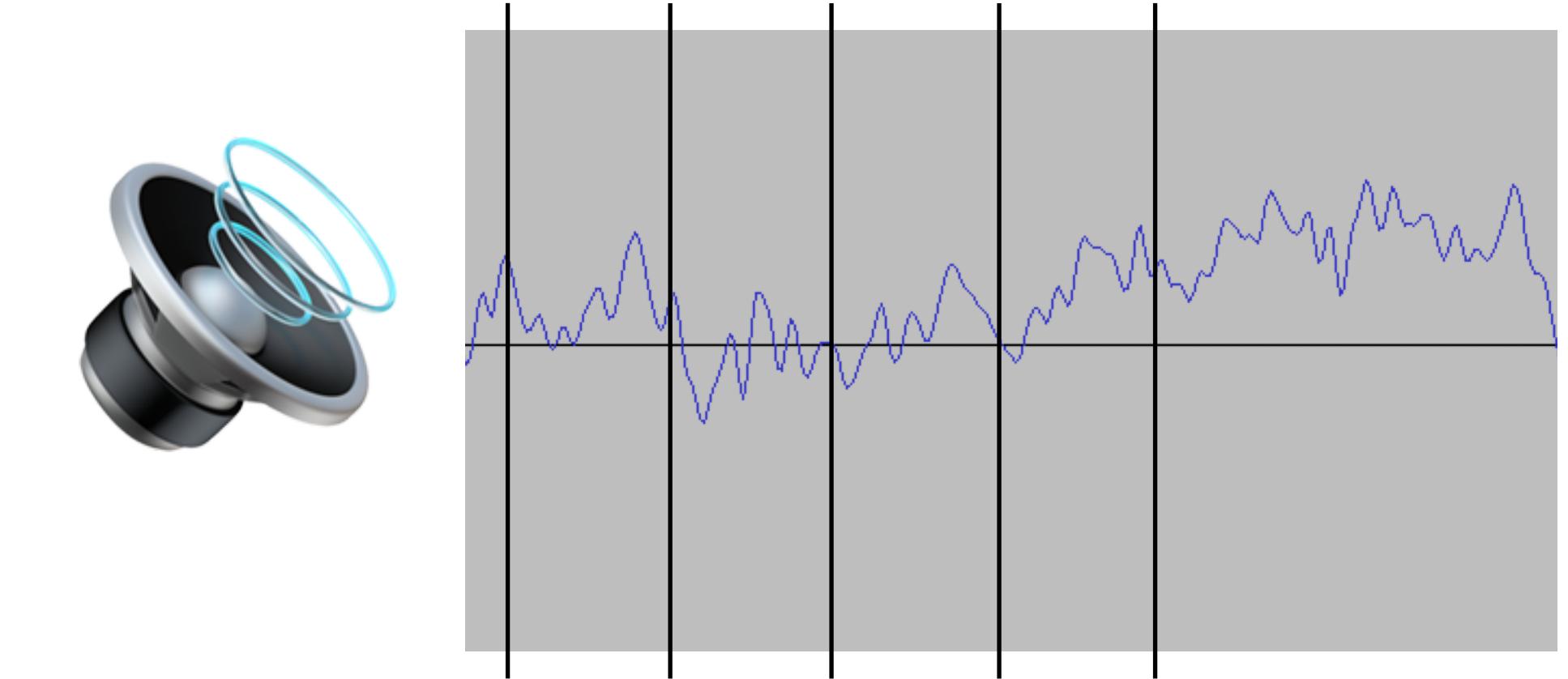
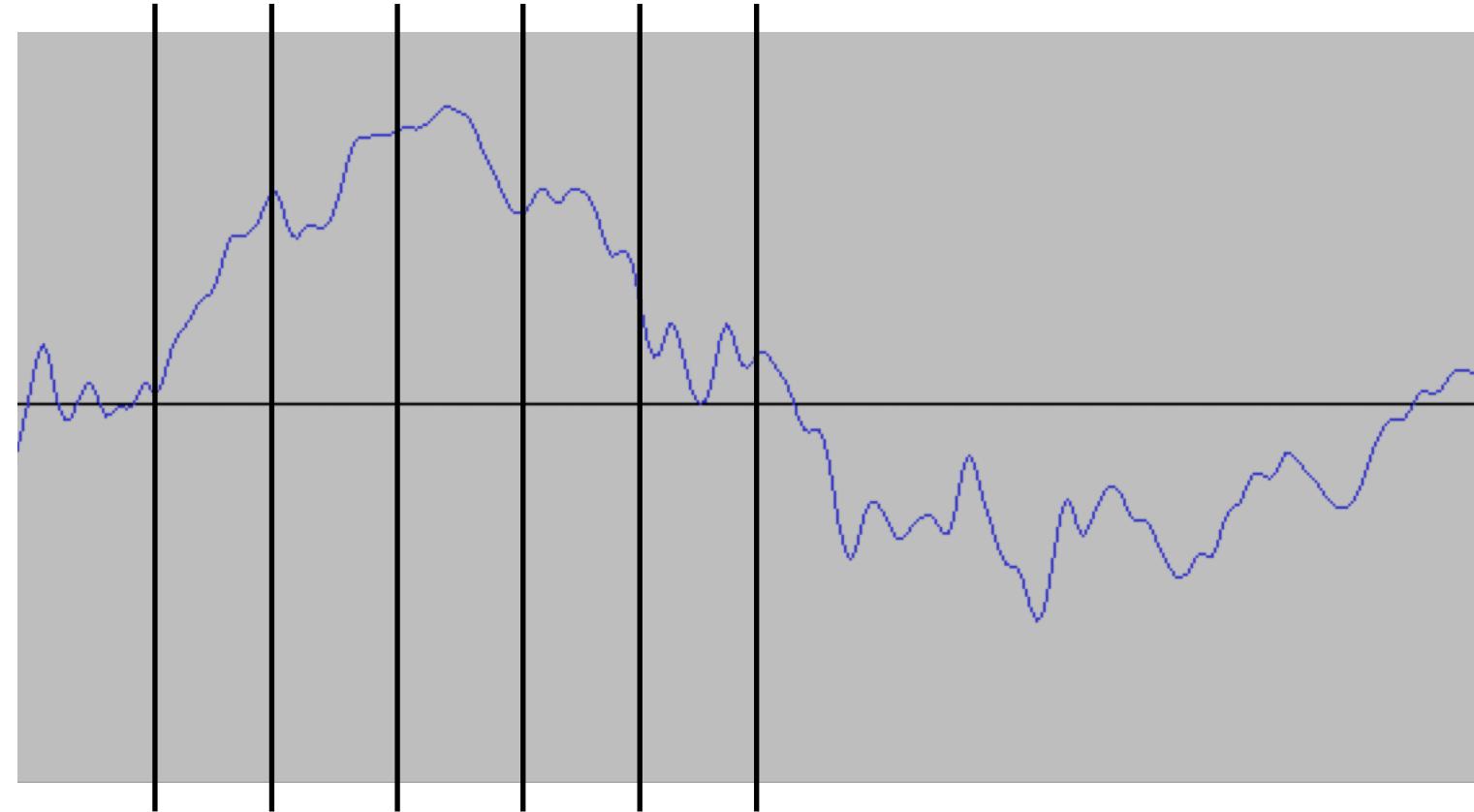
1. Overview of audio systems
2. Benchmarking & measurement
3. Optimisation
4. Multi-threading, CPUs and memory

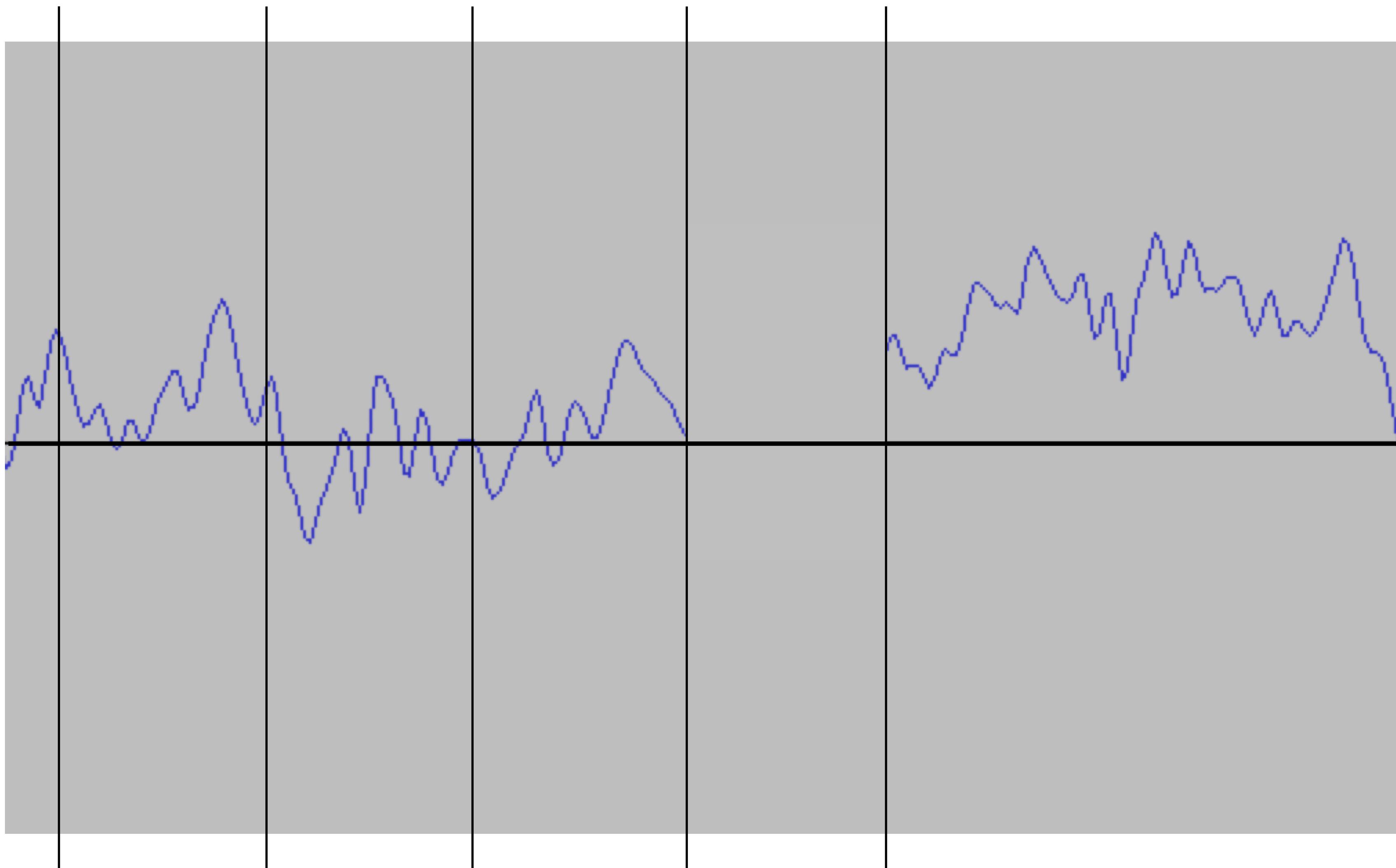
1. Overview of audio systems







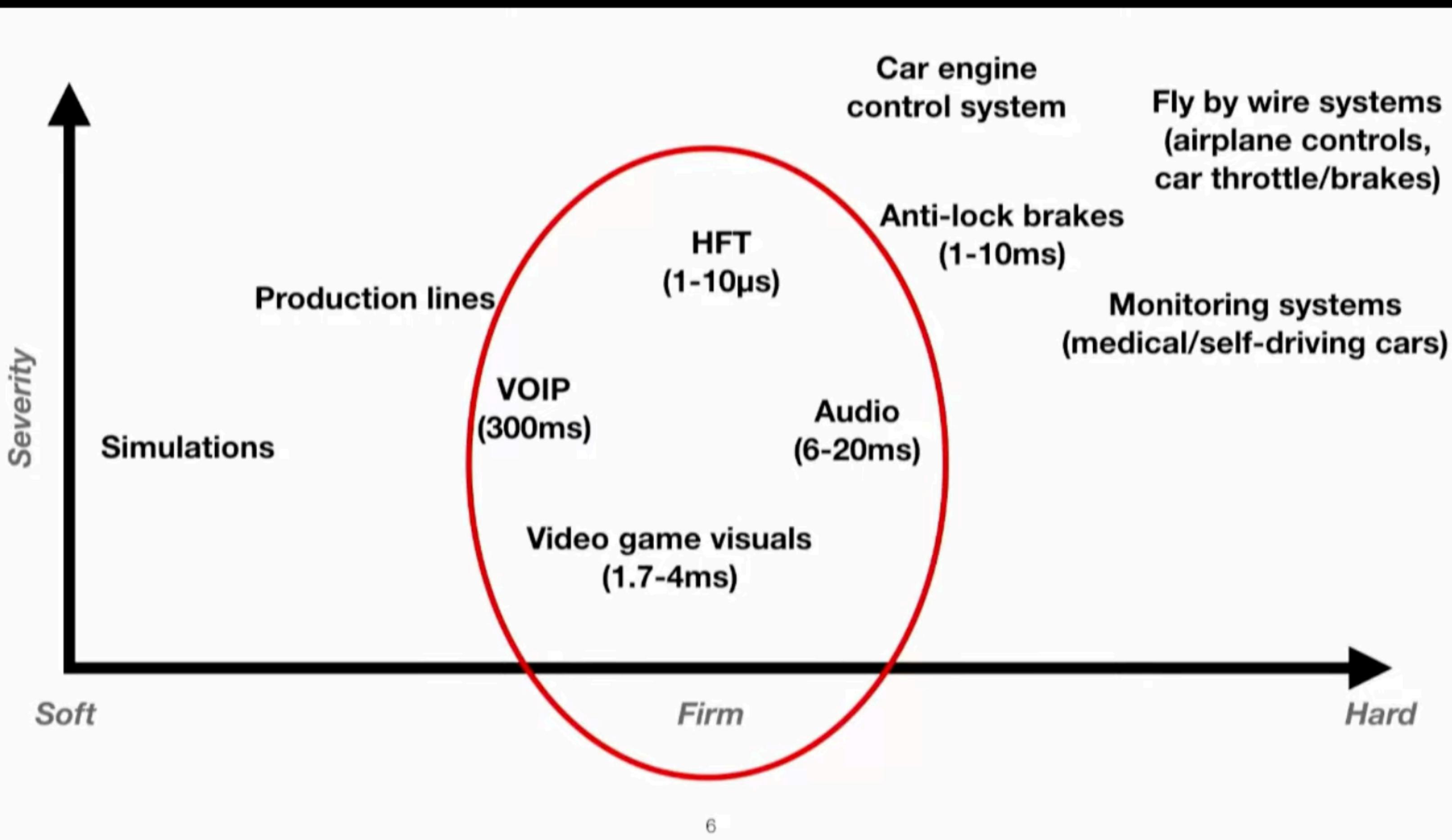






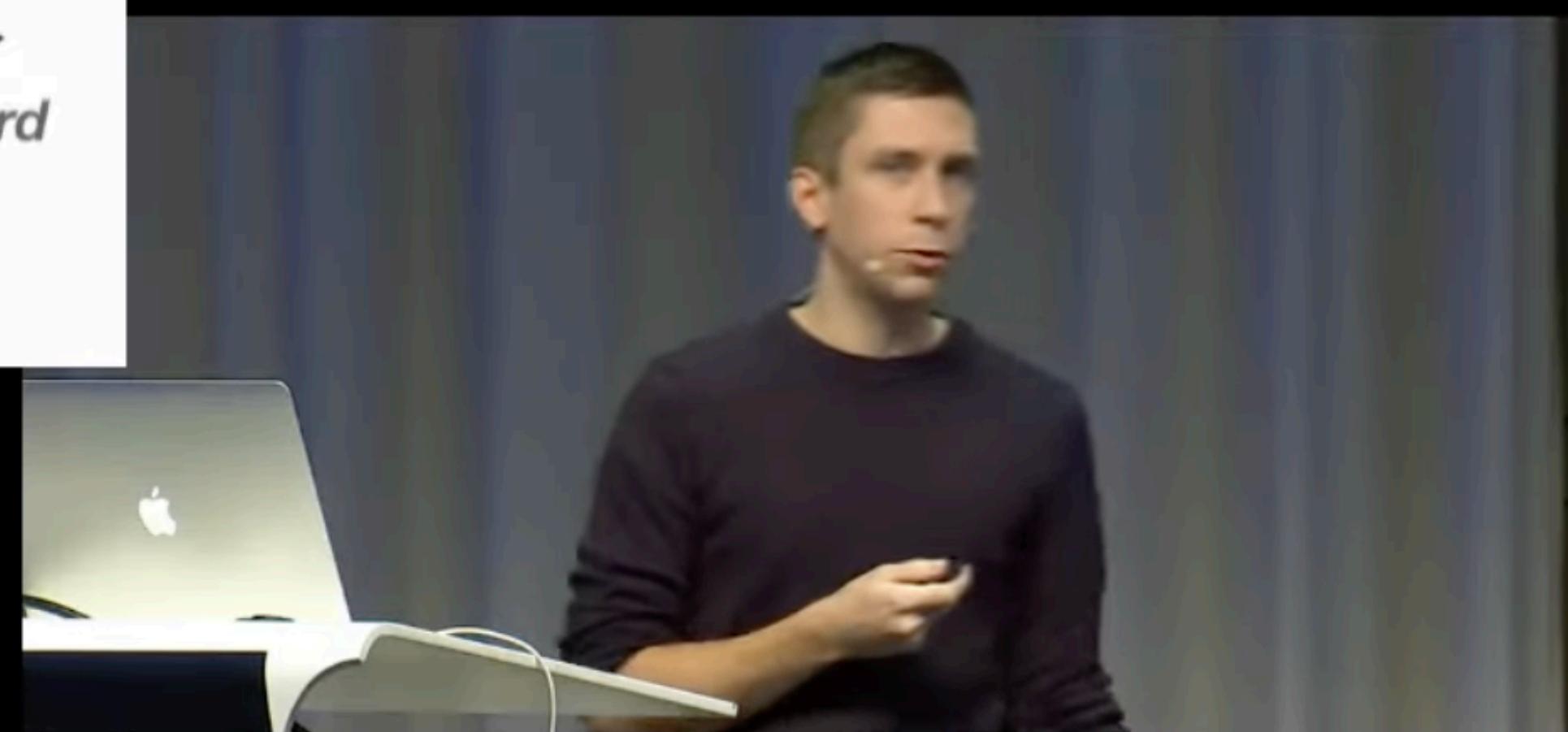


Meeting C++ 2019



**David Rowland
Fabian Renn-Giles**

Real-time 101



2. Benchmarking & Measurement

What are Benchmarks?

- Measure code execution “time”
- Assess performance characteristics
- Structured, repeatable way
- Looking for trends:
 - ***Identify regressions***
 - ***Measure optimisation attempts***
 - ***Compare environments***

Micro vs Macro Benchmarks?

- **Micro Benchmarks:**
 - Measure a single component or task
 - Low level
 - Instructions, cache misses etc.
 - Akin to unit tests
- **Macro Benchmarks:**
 - Measure a whole system
 - Higher level
 - Throughput, execution time
 - Akin to integration tests
 - Reflect what your users will see

Measurement Metrics?

- A piece of code executing many times
- Total, min/max, mean & variance/SD
- Total execution time is useful for “macro” characteristics
- Max is useful for “worst case” scenarios
- Min informs of where you could be
- Mean gives an “expected” indication

What to Record?

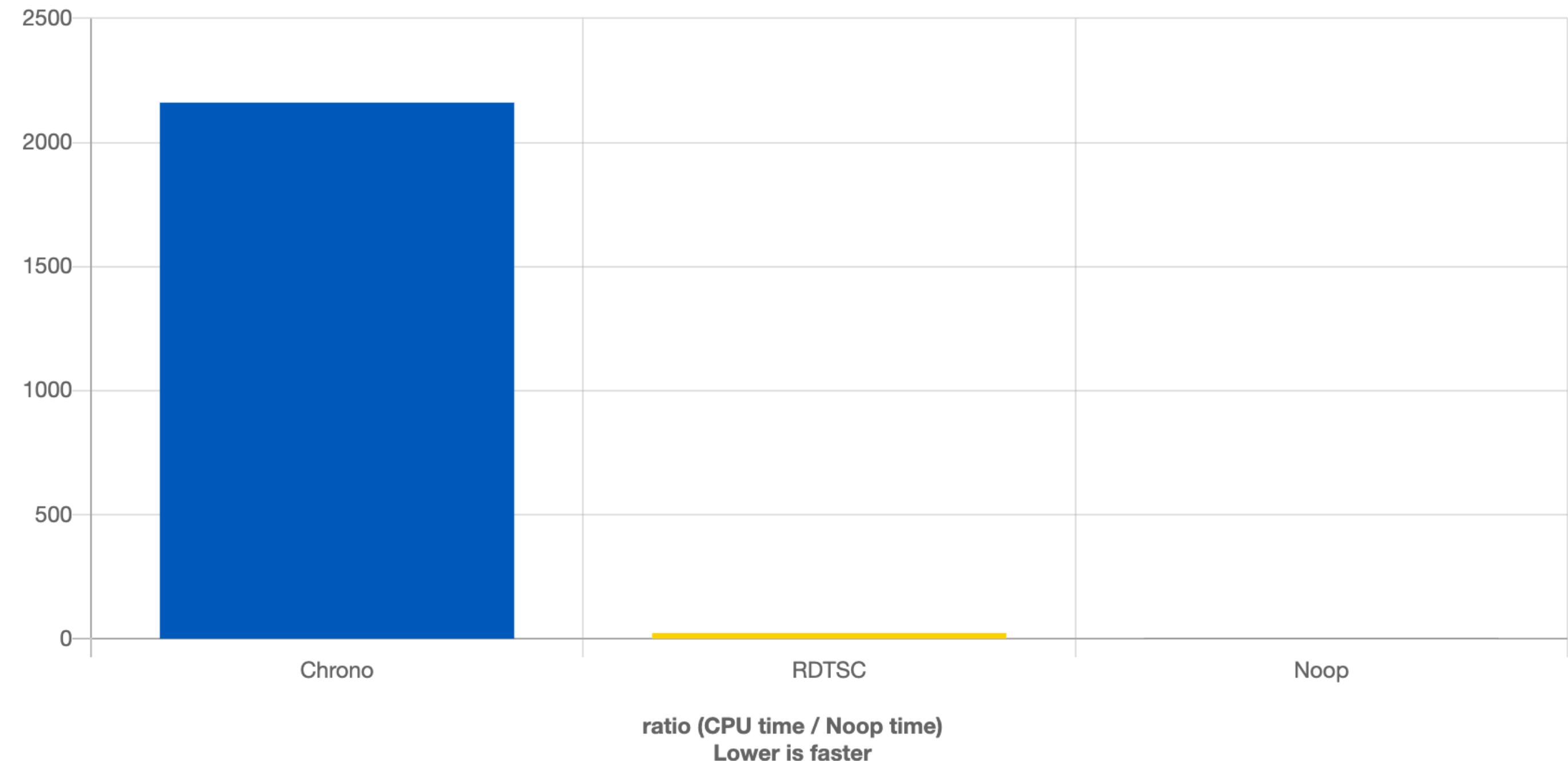
- Time
 - `chrono::time_point<chrono::high_resolution_clock>`
 - Indicates what a “typical user” will experience
 - Very environment dependant
- CPU time stamp
 - Clock cycles
 - Less overhead to measure
 - Easier to compare between platforms

quick-bench.com



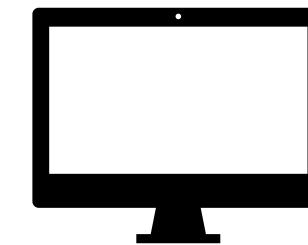
```
static void Chrono(benchmark::State& state)
{
    for (auto _ : state)
    {
        auto x = std::chrono::high_resolution_clock::now();
        benchmark::DoNotOptimize(x);
    }
}
BENCHMARK(Chrono);

static void RDTSC(benchmark::State& state)
{
    for (auto _ : state)
    {
        auto x = rdtsc();
        benchmark::DoNotOptimize(x);
    }
}
BENCHMARK(RDTSC);
```

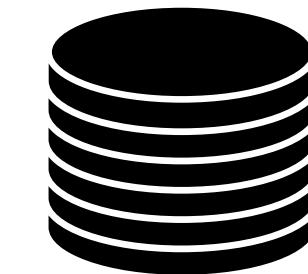


https://quick-bench.com/q/siAqlrvTsb-UB7GdpF_sJQ69v54

Where to Measure and Where to Store Results?

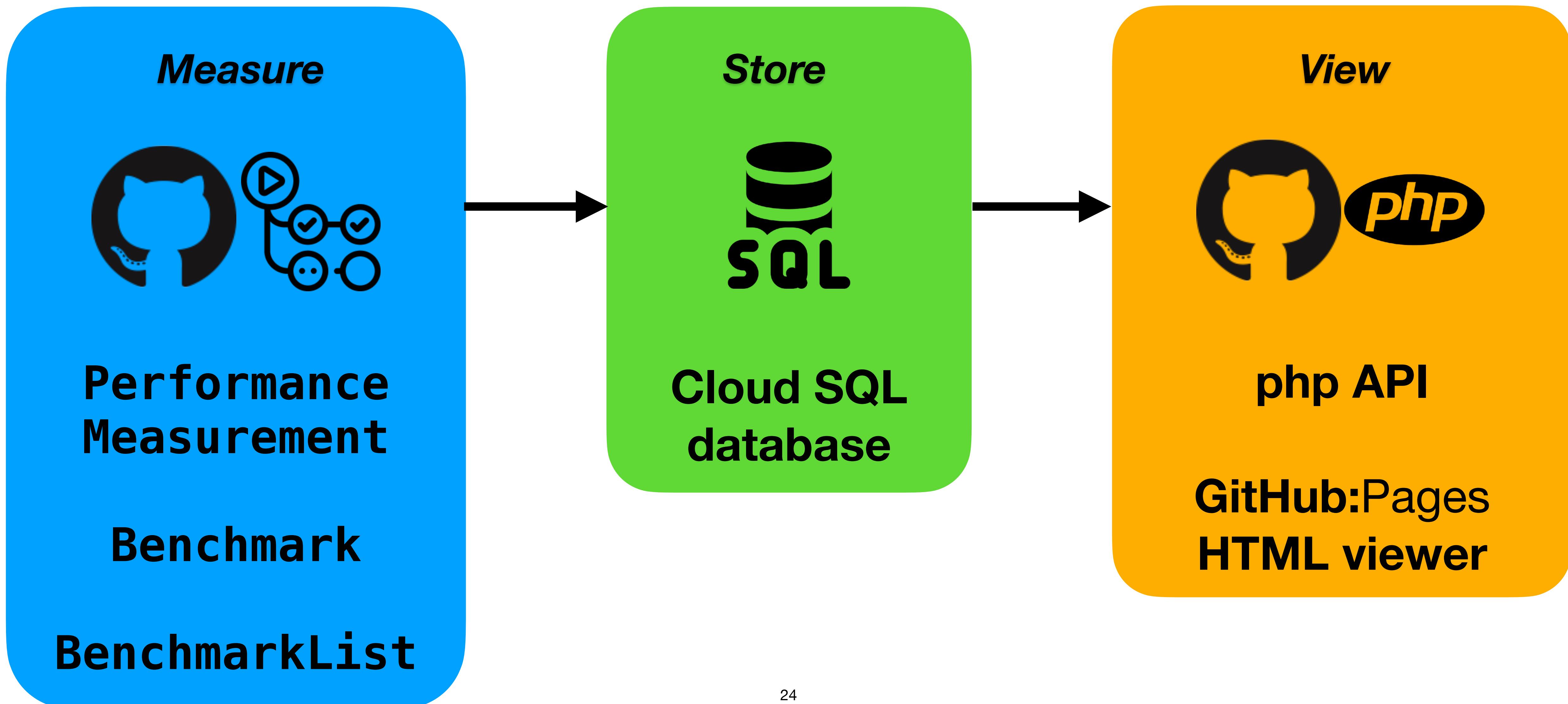


- Stable environments
 - Ideally constant machines
 - Cloud VMs provide indications



- Store results in cloud database
 - Easy to view results
 - Automatic notifications?

Benchmark System Diagram



```

//=====
class PerformanceMeasurement
{
public:
//=====
/** Creates a PerformanceMeasurement object.

    @param counterName      the name used when printing out the statistics
    @param runsPerPrintout  the number of start/stop iterations before calling
                           printStatistics()

*/
PerformanceMeasurement (const std::string& counterName,
                       int runsPerPrintout = 100,
                       bool printOnDestruction = true);

/** Destructor.*/
~PerformanceMeasurement();

//=====
/** Starts timing.
    @see stop
*/
void start() noexcept;

/** Stops timing and prints out the results.

    The number of iterations before doing a printout of the
    results is set in the constructor.

    @see start
*/
bool stop();

/** Dumps the current metrics to std::cout.*/
void printStatistics();

/** Returns a copy of the current stats.*/
Statistics getStatistics() const;

```

```

/** Holds the current statistics.*/
struct Statistics
{
    Statistics() noexcept = default;

    void clear() noexcept;
    double getVarianceSeconds() const;
    double getVarianceCycles() const;
    std::string toString() const;

    void addResult (double secondsElapsed, uint64_t cyclesElapsed);

    std::string name;

    double meanSeconds      = 0.0;
    double m2Seconds        = 0.0;
    double maximumSeconds   = 0.0;
    double minimumSeconds   = 0.0;
    double totalSeconds     = 0.0;

    double meanCycles       = 0.0;
    double m2Cycles          = 0.0;
    uint64_t maximumCycles  = 0;
    uint64_t minimumCycles  = 0;
    uint64_t totalCycles     = 0;

    int64_t numRuns = 0;
};

```

```

/** Describes a benchmark.
These fields will be used to sort and group your benchmarks for comparison over time.
*/
struct BenchmarkDescription
{
    size_t hash = 0;          /**< A hash uniquely identifying this benchmark. */
    std::string category;    /**< A category for grouping. */
    std::string name;        /**< A human-readable name for the benchmark. */
    std::string description; /**< An optional description that might include configs etc. */
    std::string platform { juce::SystemStats::getOperatingSystemName().toString() };
};

/** Holds the duration a benchmark took to run.*/
struct BenchmarkResult
{
    BenchmarkDescription description;
    MeasurementResults metrics;
    juce::Time date { juce::Time::getCurrentTime() };
};

```

```

//=====
//**
// An individual Benchmark.
// To measure a benchmark, simply create one of these with a valid description
// then before the code you are measuring call start and stop afterwards.
// Once you've done that, call getResult() to return the duration the benchmark took to run.
//
// To collect a set of BenchmarkResults see @BenchmarkList
*/
class Benchmark
{
public:
    /** Creates a Benchmark for a given BenchmarkDescription. */
    Benchmark (BenchmarkDescription desc)
        : description (std::move (desc))
    {}

    /** Starts timing the benchmark. */
    void start()
    {
        measurement.start();
    }

    /** Stops timing the benchmark. */
    void stop()
    {
        measurement.stop();
    }

    /** Returns the timing results. */
    BenchmarkResult getResult() const
    {
        return createBenchmarkResult (description, measurement.getStatistics());
    }

private:
    BenchmarkDescription description;
    traktion::graph::PerformanceMeasurement measurement { {}, -1, false };
};

```

```

class ResamplingBenchmarks : public juce::UnitTest
{
public:
    ResamplingBenchmarks()
        : juce::UnitTest ("Resampling Benchmarks", "tracktion_benchmarks")
    {}

void runTest() override
{
    runResamplingRendering ("lagrange",      ResamplingQuality::lagrange);
    runResamplingRendering ("sincFast",       ResamplingQuality::sincFast);
    runResamplingRendering ("sincMedium",     ResamplingQuality::sincMedium);
    runResamplingRendering ("sincBest",       ResamplingQuality::sincBest);
}

private:
//=====
//=====

void runResamplingRendering (juce::String qualityName,
                             ResamplingQuality quality)
{

```

Create a 30s sin clip

```

waveClip->setResamplingQuality (quality);

{
    ScopedBenchmark sb (createBenchmarkDescription ("Resampling", "WaveNode quality", "30s sin wave, 96KHz to 44.1Khz, " + qualityName.toStdString()));
    Renderer::measureStatistics ("Rendering resampling",
                                *edit, timeRange,
                                toBitSet ({ t }),
                                256, playbackSampleRate);
}

};

static ResamplingBenchmarks resamplingBenchmarks;

```

Tracktion Benchmarks

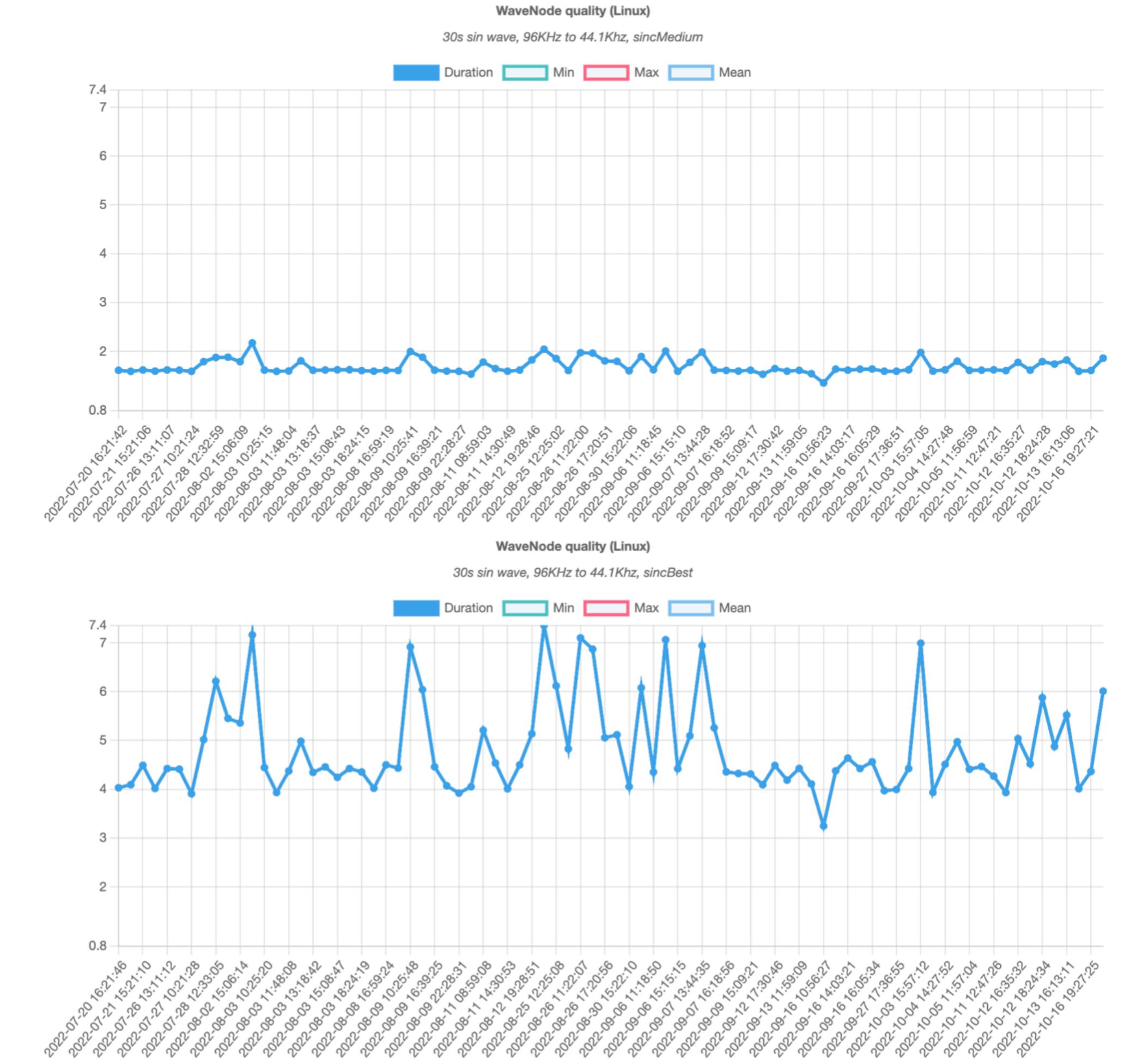
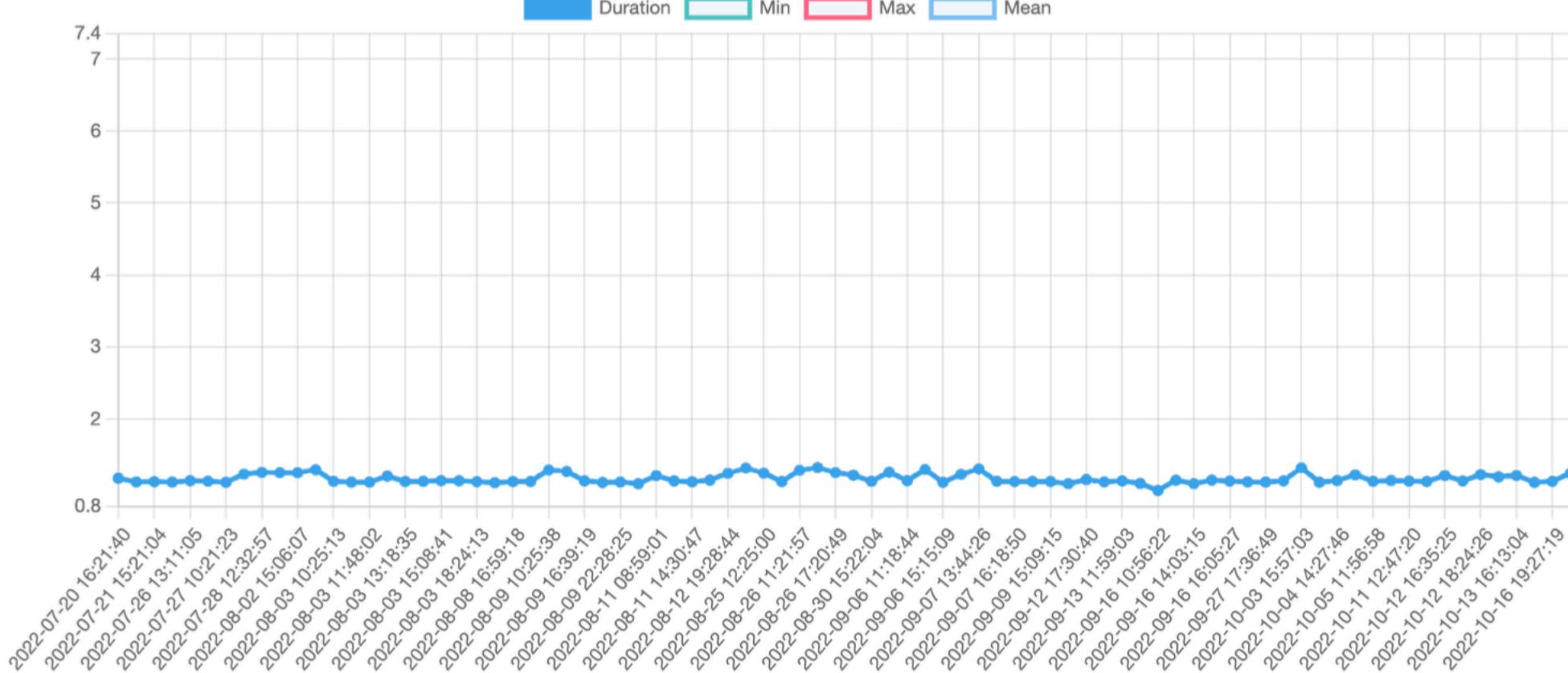
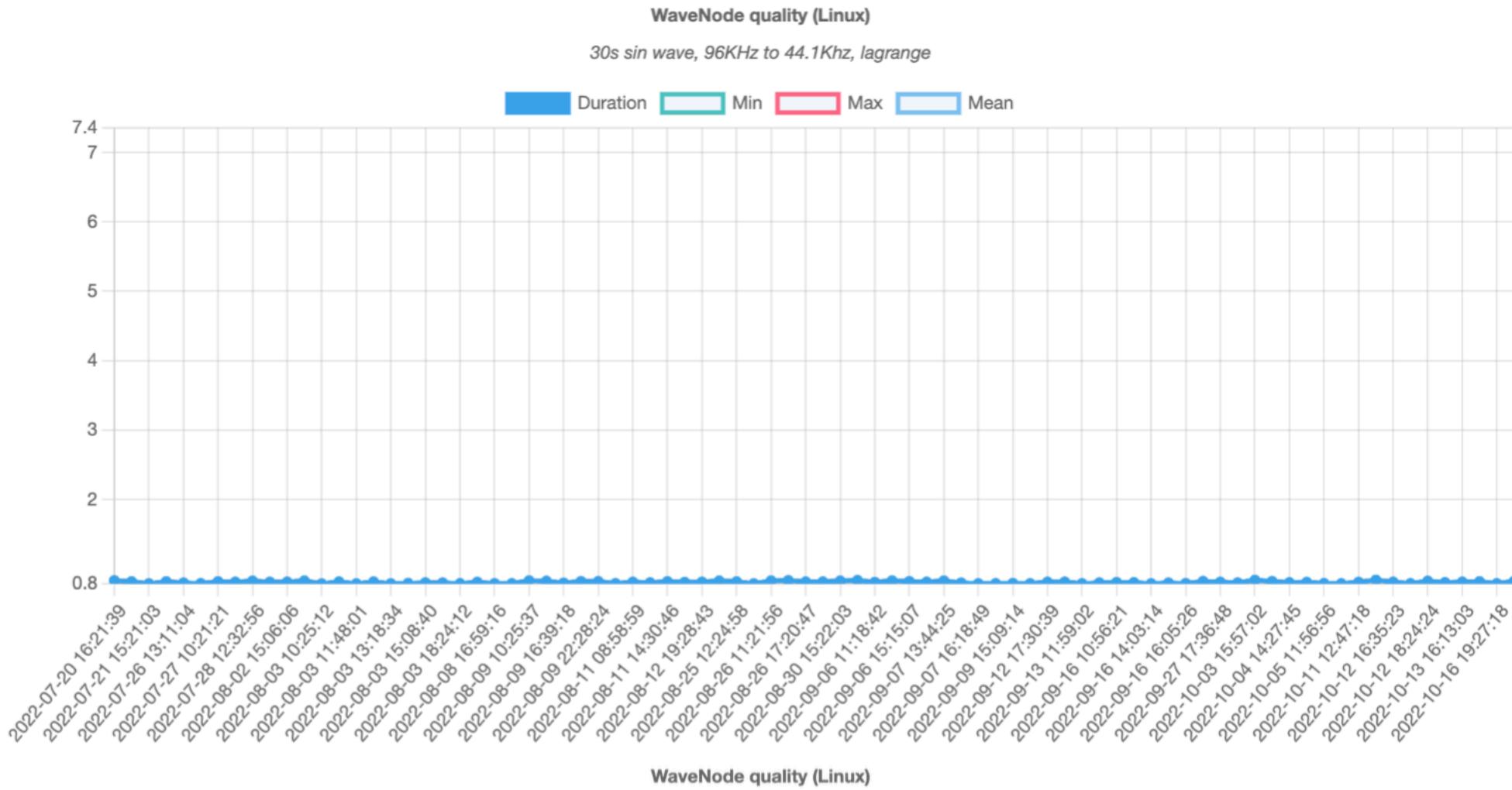
Categories: All ▼ Name: WaveNode quality ▼ Description:

All ▼ Platform: Linux ▼

Start Date: dd/mm/yyyy End Date: dd/mm/yyyy Time: Cycles

Show total Show min/max Show variance Normalise results

Update Results

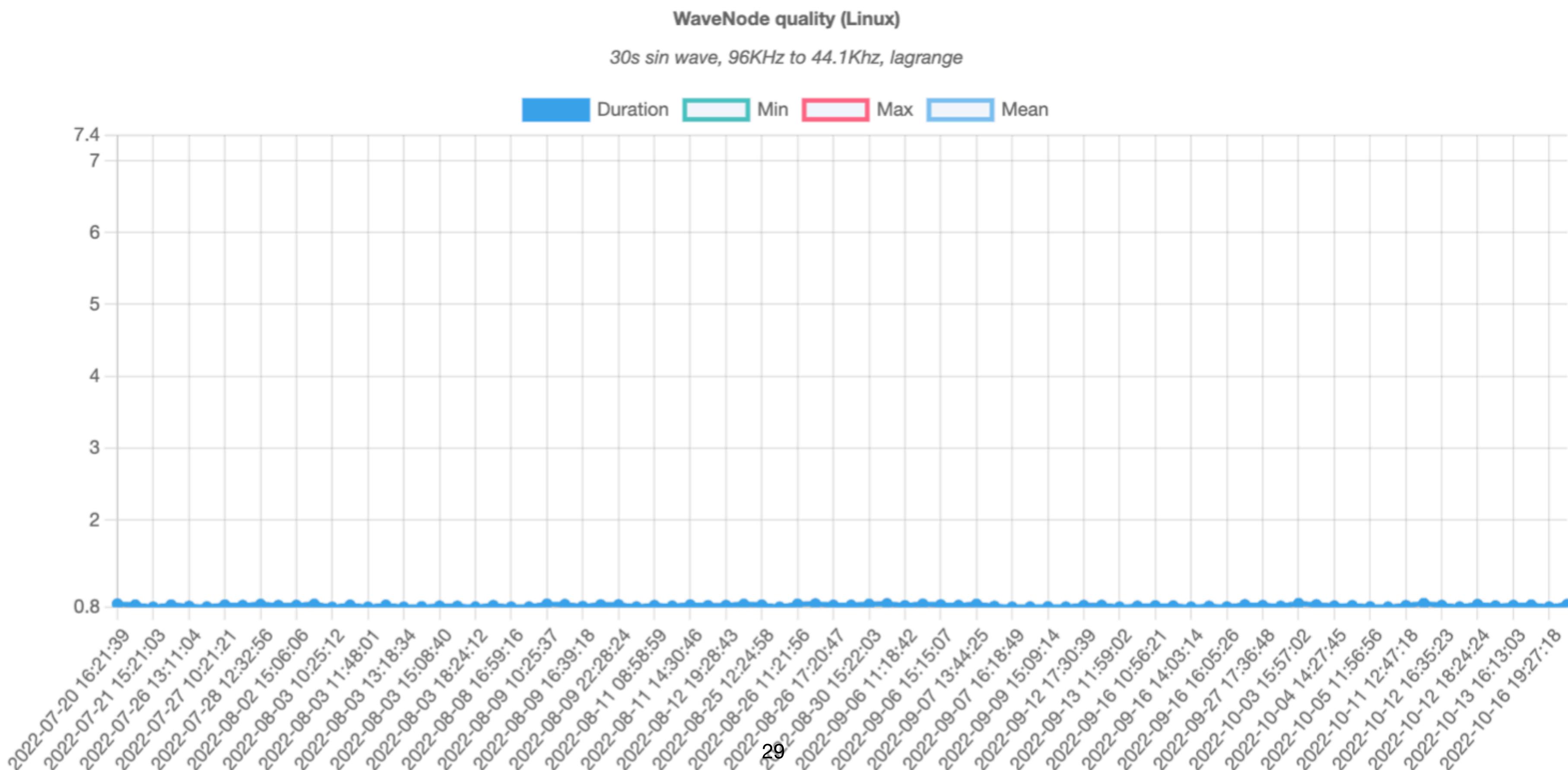


Tracktion Benchmarks

Categories: All Name: WaveNode quality Description:
All Platform: Linux

Start Date: dd/mm/yyyy End Date: dd/mm/yyyy Time Cycles

Show total Show min/max Show variance Normalise results



Tracktion Benchmarks

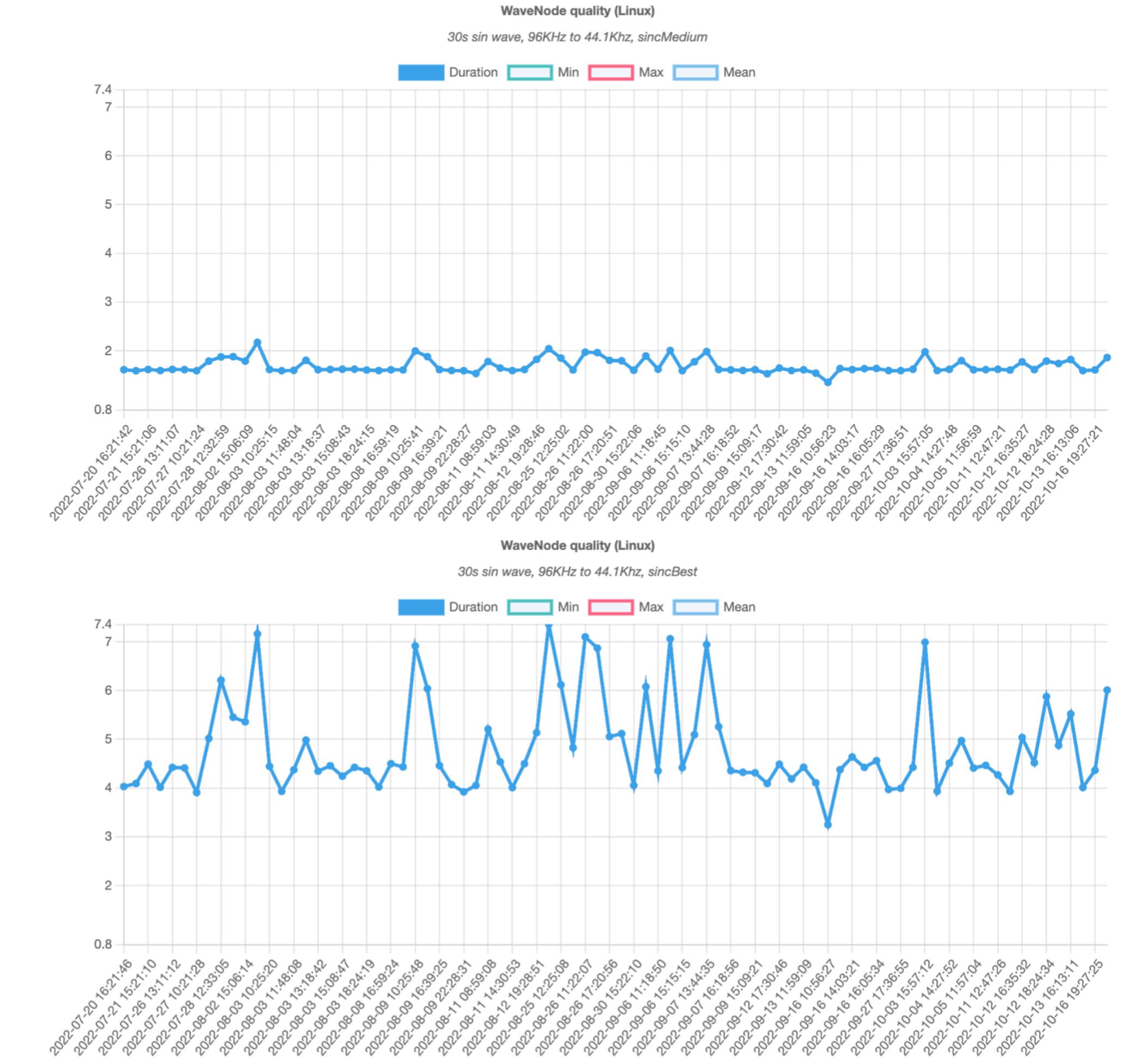
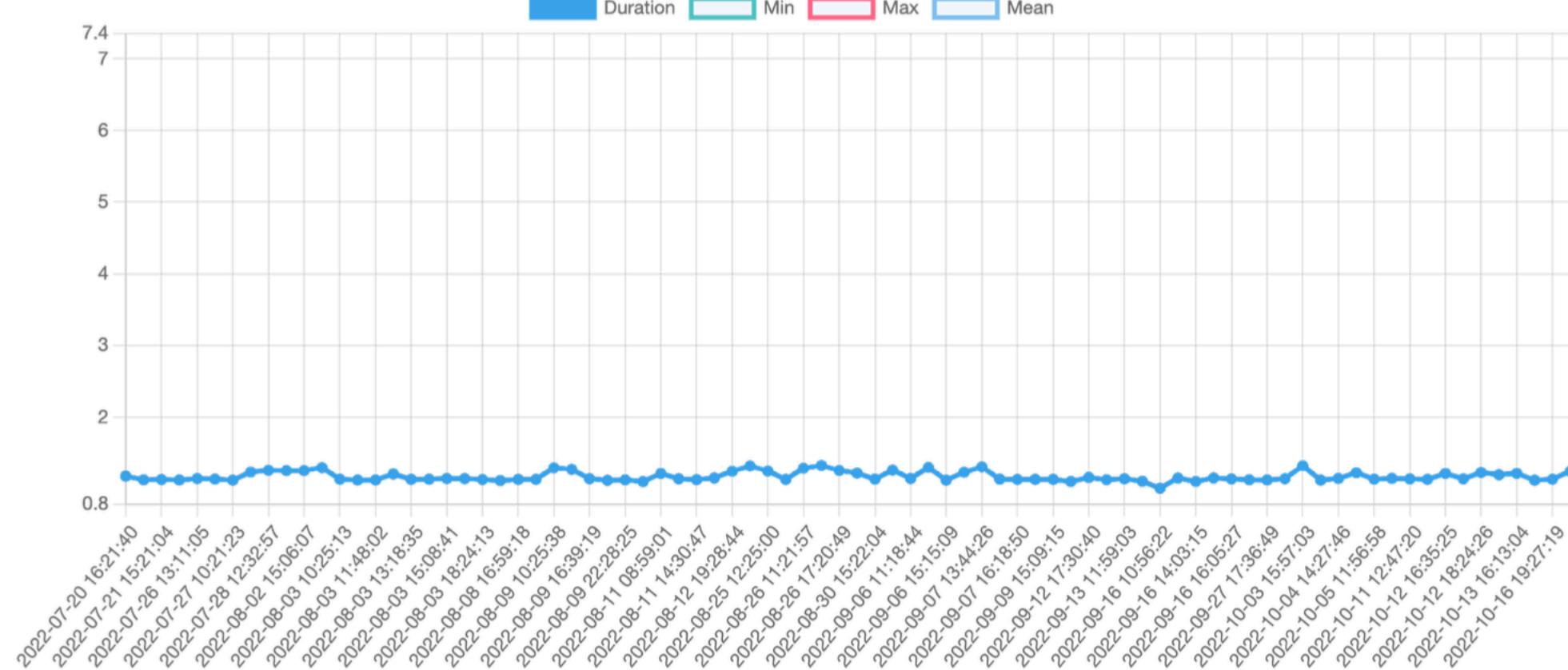
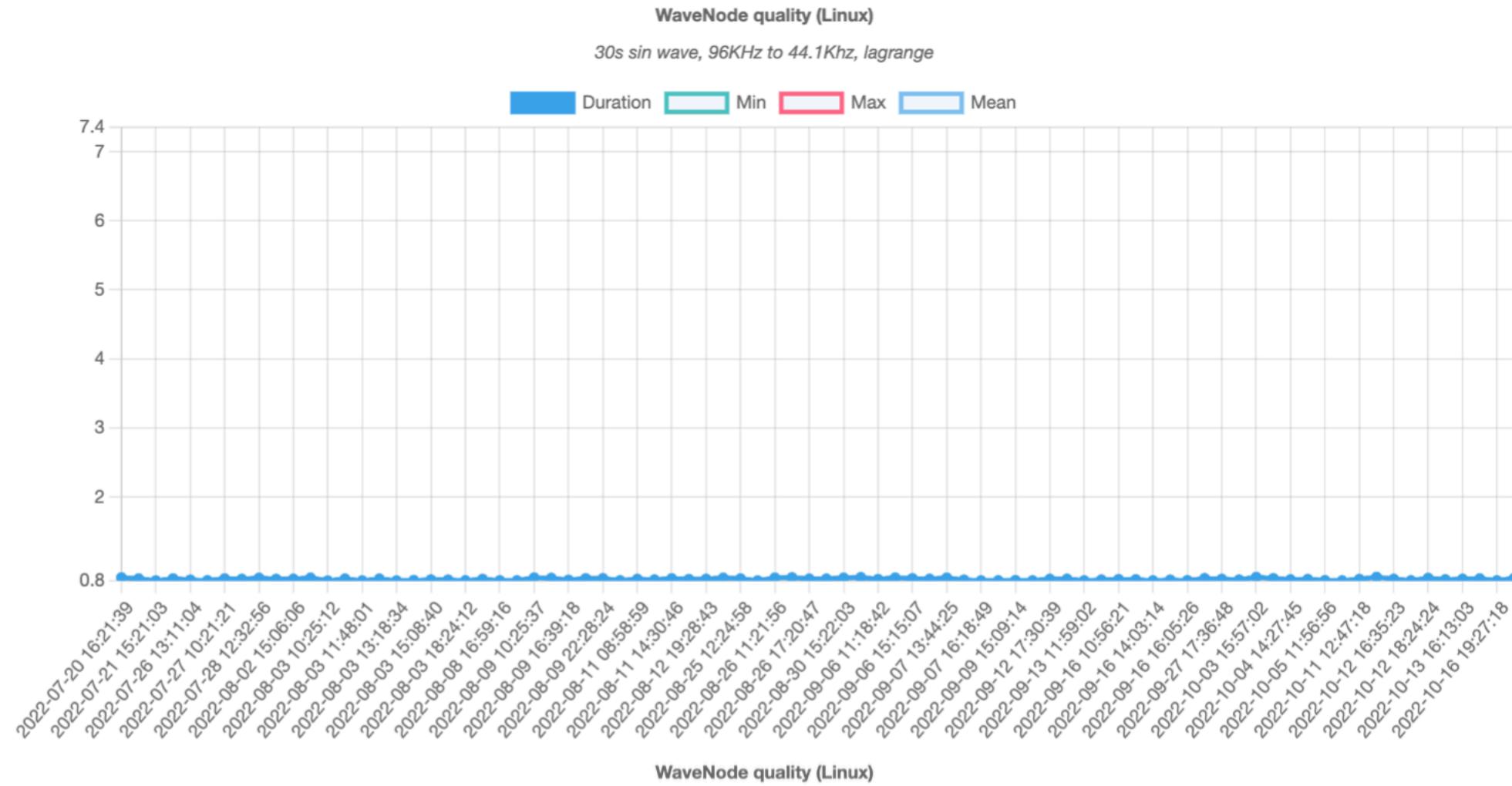
Categories: All ▼ Name: WaveNode quality ▼ Description:

All ▼ Platform: Linux ▼

Start Date: dd/mm/yyyy End Date: dd/mm/yyyy Time Cycles

Show total Show min/max Show variance Normalise results

Update Results



When to add benchmarks to your suite?

- New features
- QA/user reports
- When you see benchmark regressions

3. Optimisation

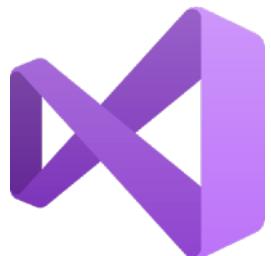
Identifying Areas for Optimisation

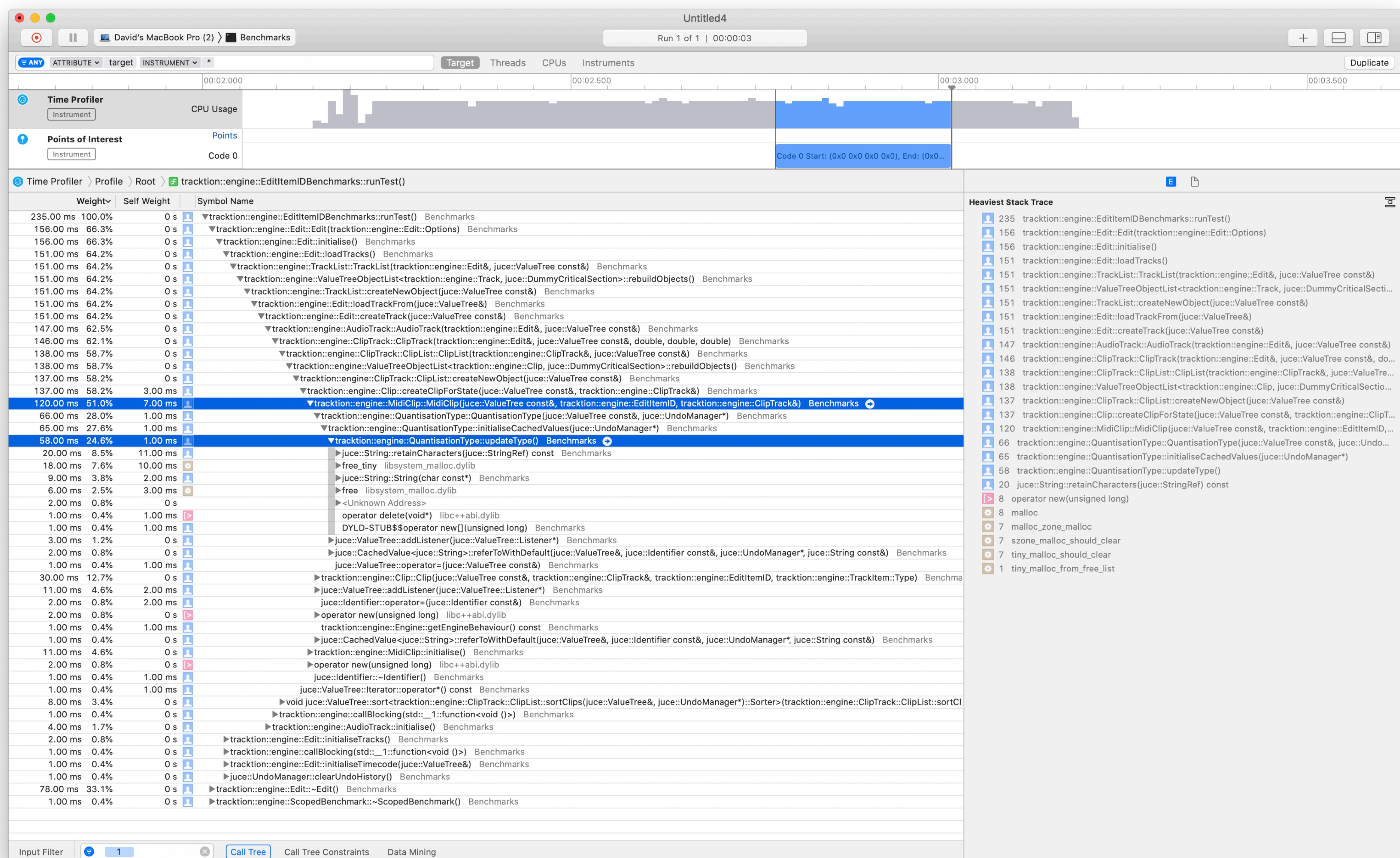
- Run a specific benchmark under a tool to get a picture of what is going on



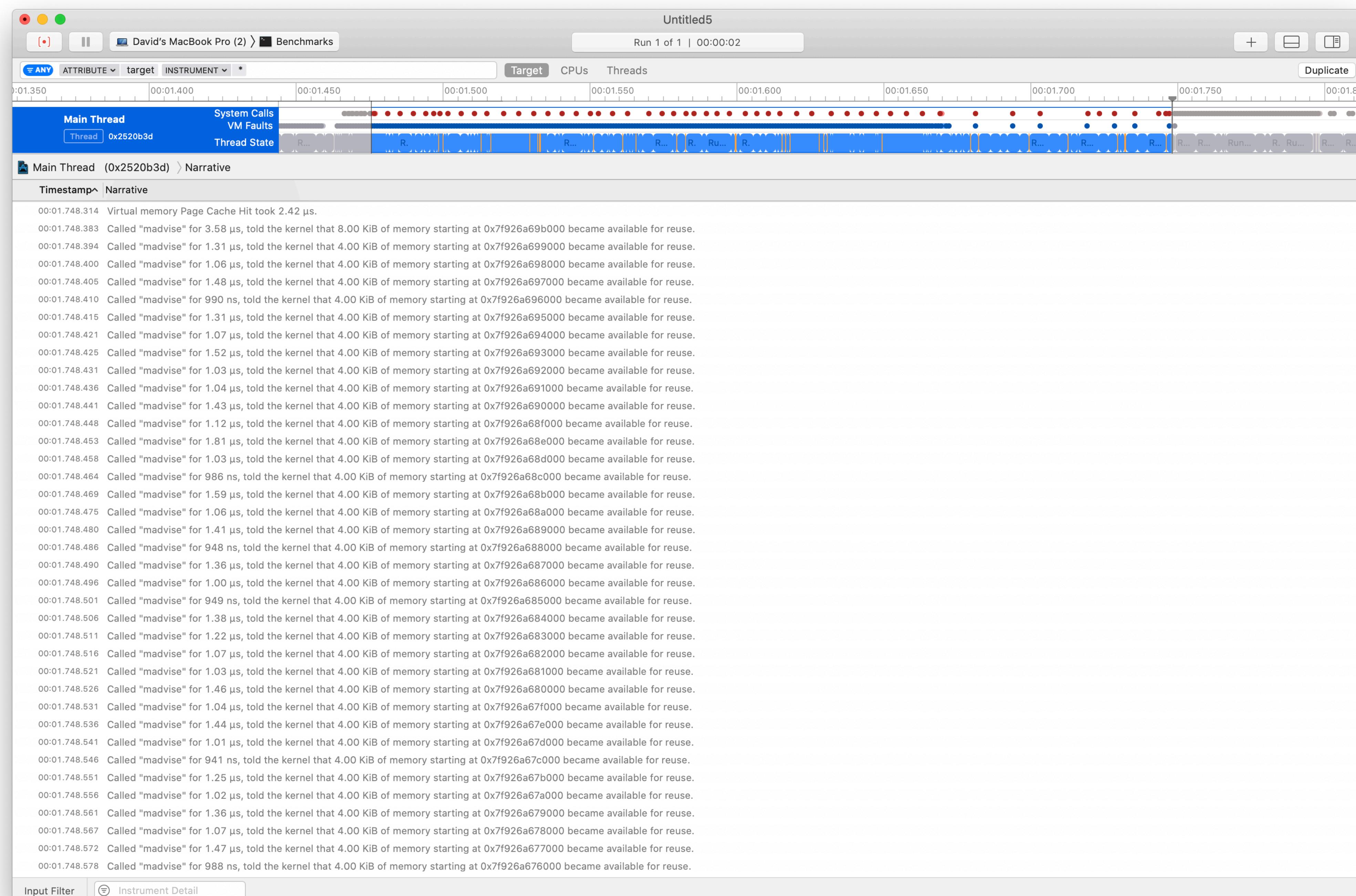
Profiling Tools

- Time profilers:
 - Give you a tree of time spent in functions
 - Can be used to identify hotspots
 - Xcode Instruments - Time Profiler, VS Profile, Intel VTune
- System Tracers:
 - Show events such as system calls, memory management, thread interruptions etc.
 - Xcode Instruments - System Trace, DTrace/strace
- Performance counters:
 - Reads CPU hardware registers for things like branch mis-predictions, cache misses etc.
 - Xcode Instruments - Counters, Linux - perf
- **Always profile release builds!**

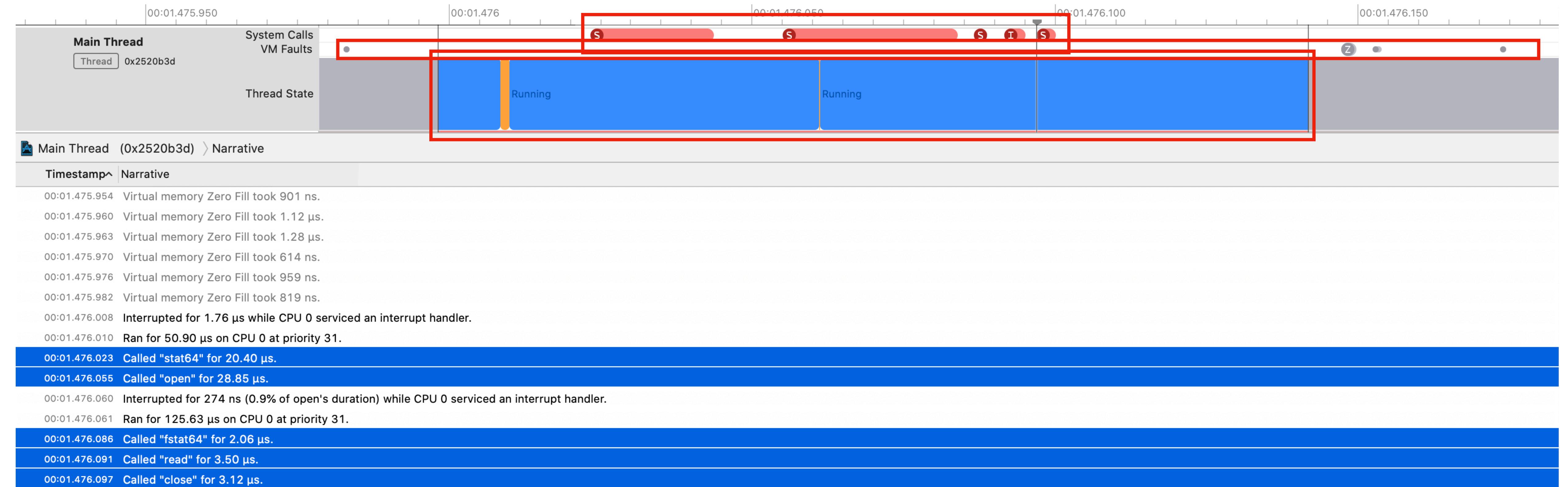


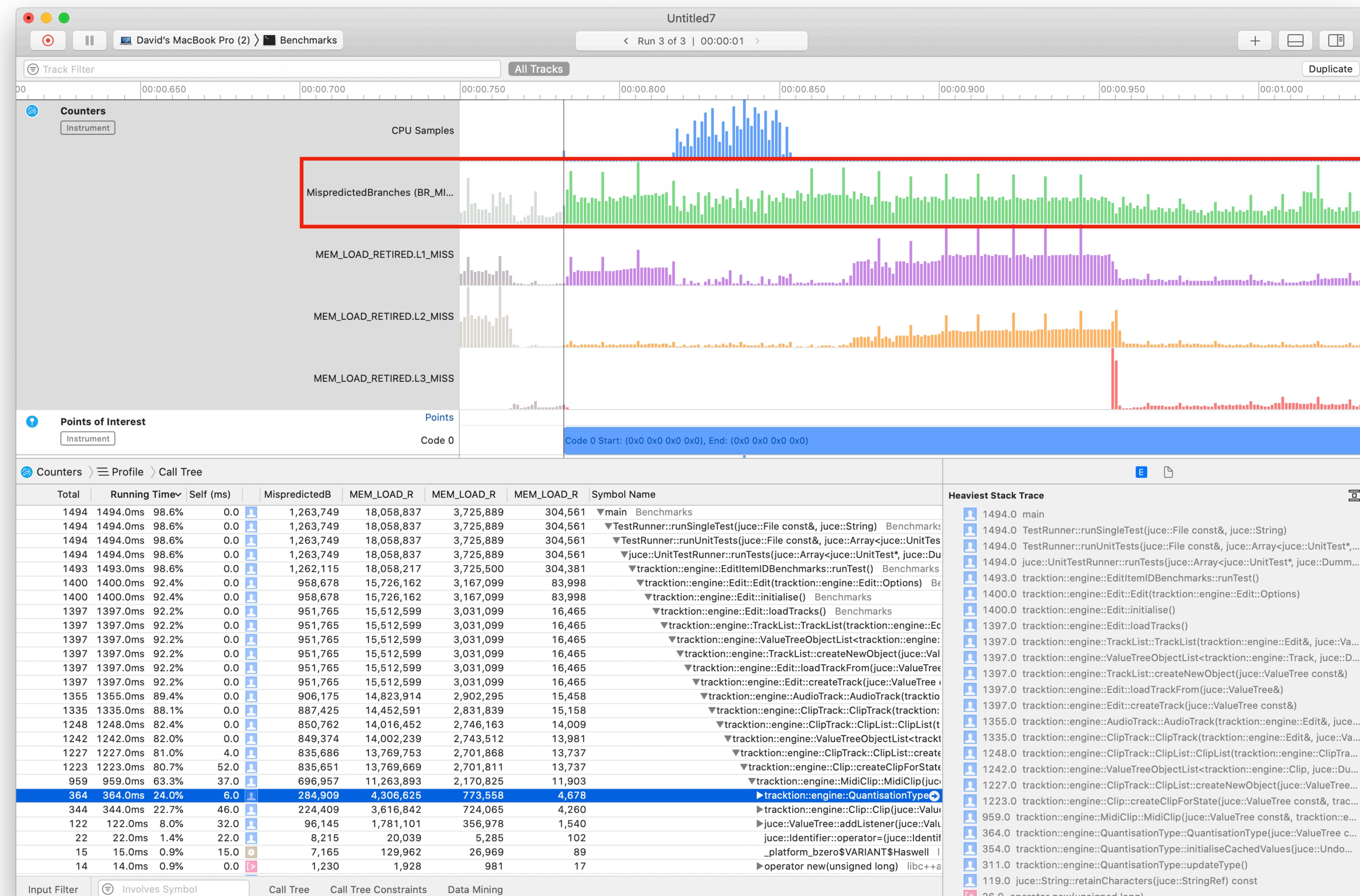


Time Profiler



System Trace





Performance Counters

Memory Access







Techniques for Optimisation

Techniques for Optimisation

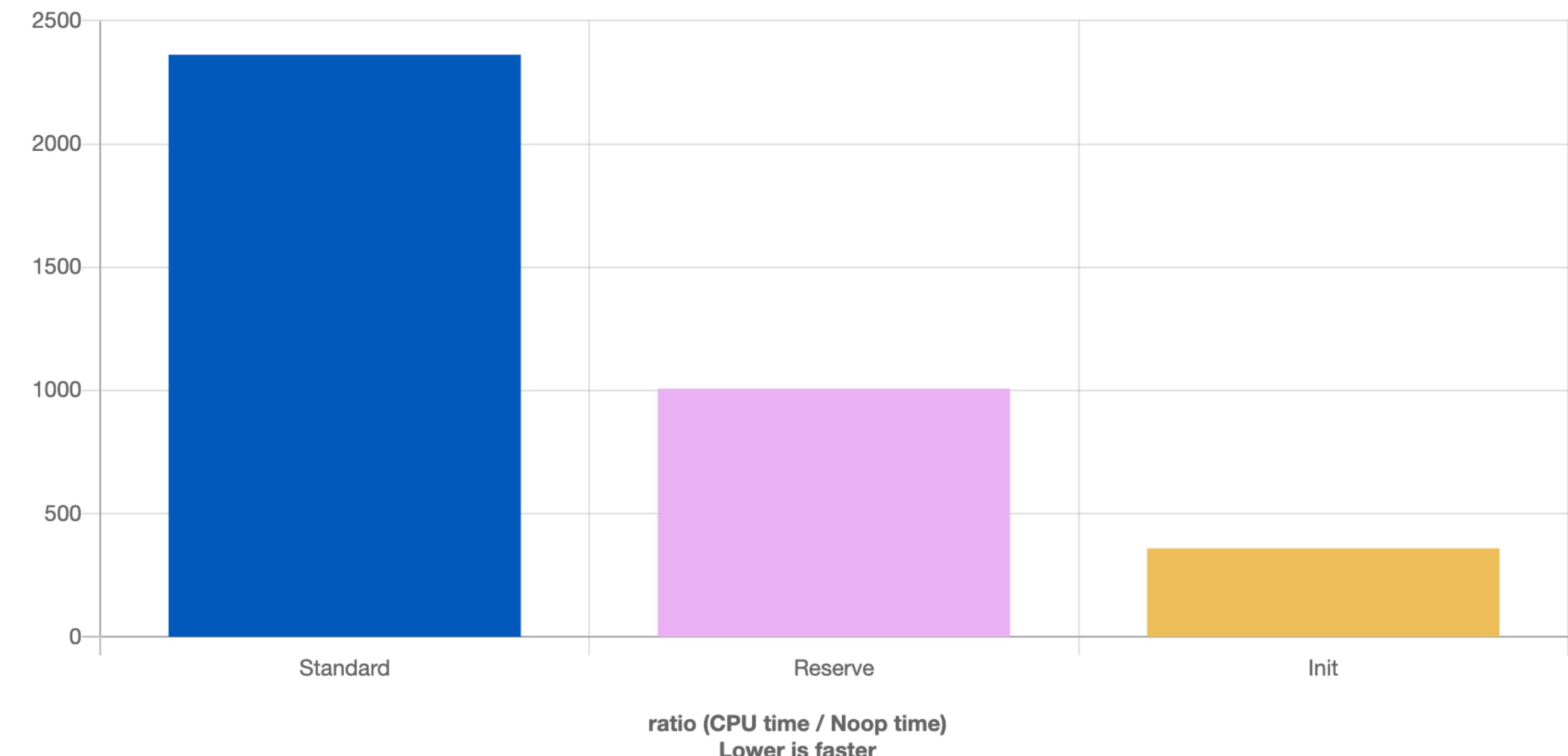
1. Identifying work that can be removed

- The simplest form of optimisation is not doing some work
- Use a profile trace to identify sections of code that don't need to be run
 - E.g. reducing the number of memory allocations by reserving
- You may need to think of an alternative approach in order to remove a chunk of code
 - Evaluate lazily
 - Evaluate asynchronously

```

1 #include <vector>
2
3
4
5
6
7     std::vector<double> vec;
8
9     for (int i = 0; i < 1'000; ++i)
10    |   vec.push_back (0.0);
11
12
13
14
15
16
17
18     std::vector<double> vec;
19     vec.reserve (1'000);
20
21     for (int i = 0; i < 1'000; ++i)
22    |   vec.push_back (0.0);
23
24
25
26
27
28
29
30
31
32     std::vector<double> vec (1'000, 0.0);
33
34
35
36

```



Techniques for Optimisation

2. Identifying work that can be combined

- Combine work in time:
 - Evaluate in parallel
 - Execute using SIMD instructions (single instruction, multiple data)
- Combine work in an algorithm:
 - `std::min_element + std::max_element ≈ std::minmax_element`

Credit to Fabian:

<https://forum.juce.com/t/simdregister-is-it-worth-it/53362/4>



fr810

1 7d

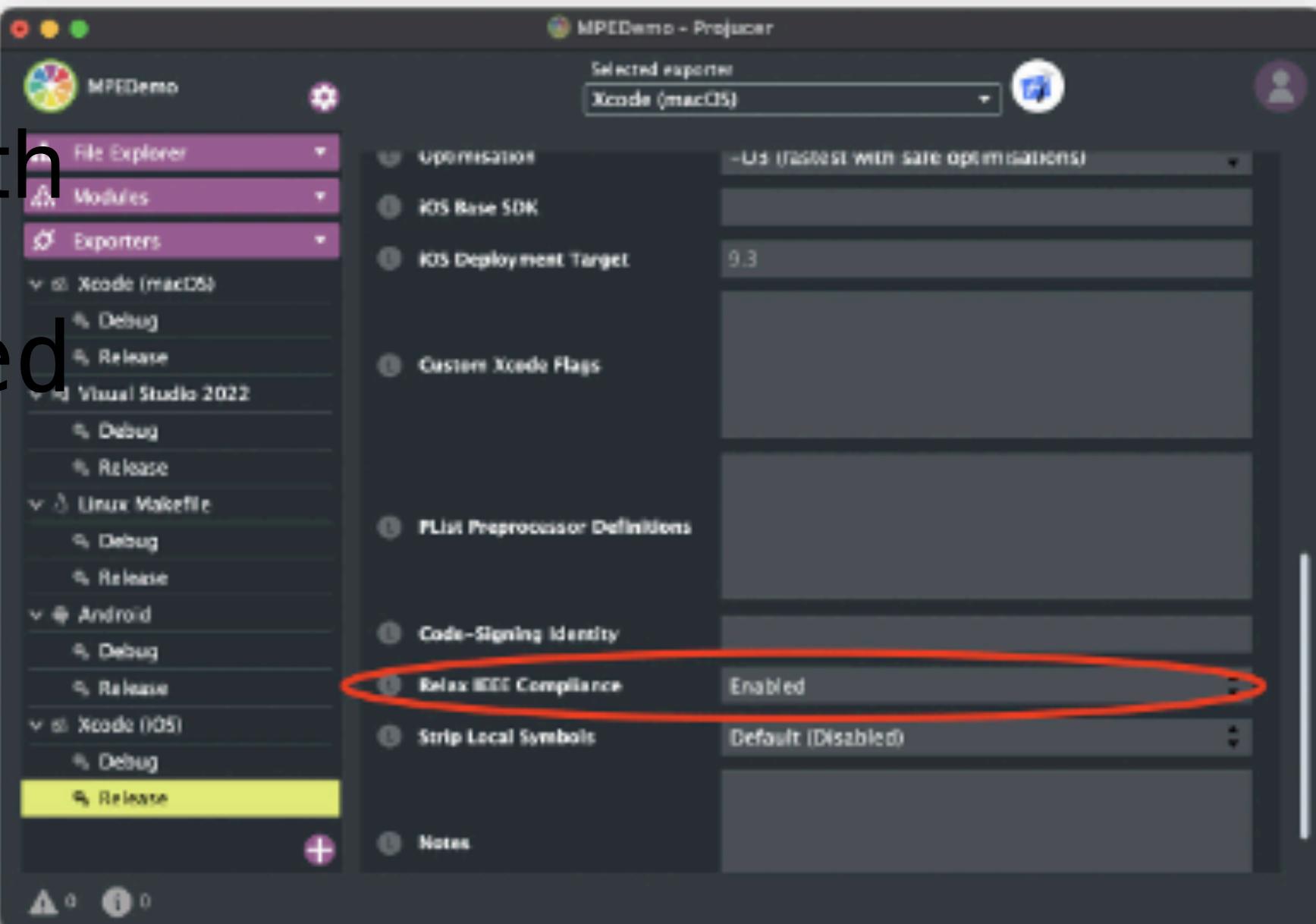
Author of SIMDRegister.h: "I'd just like to add that before embarking on SIMDRegister, check that your compiler isn't already auto-vectorizing the tight loops in your code. In my experience, the compiler does a good job auto-vectorizing even moderately complex loops (but see my note at the end of this post)."

- Build with at least -O3

To ensure that the compiler is allowed to even auto-vectorize your code, be sure that:

1. You are building in release mode (i.e. at least optimisation level -O3)
2. You have "Relax IEEE compliance" enabled in the Projucer. This is absolutely crucial, especially on arm/arm64 as SIMD instructions do not have the same denormal/round-to-zero (arm) and/or multiply-accumulate rounding (x86/arm) behaviour as normal IEEE compliant floating point instructions. Hence, the compiler is not allowed to replace your loops with SIMD instructions if it needs to ensure IEEE compliance.

- Relax IEEE compliance (`-ffast-math`)
 - `-Ofast`
- Optionally help compiler with
 - `std::assumed_aligned`



Note if you are compiling with `-Ofast` then "Relax IEEE compliance" will be enabled regardless of the Projucer setting.

To check if your code is being auto-vectorized, you need to look at the assembly. I recommend doing this, even when using SIMDRegister, as you will often have unexpected results.

```
#include <iostream>
#include <numeric>
#include <cmath>

int main()
{
    double a = std::numeric_limits<double>::infinity();

    if (std::isinf(a))
        std::cout << "Inf detected.\n";
    else
        std::cout << "Inf NOT detected.\n";

    if (std::isfinite(a))
        std::cout << "Is finite.\n";
    else
        std::cout << "Is NOT finite.\n";

    return 0;
}
```

Clang trunk: -std=c++2b -O3

Inf detected.
Is NOT finite.

-ffast-math

Inf NOT detected.
Is finite.

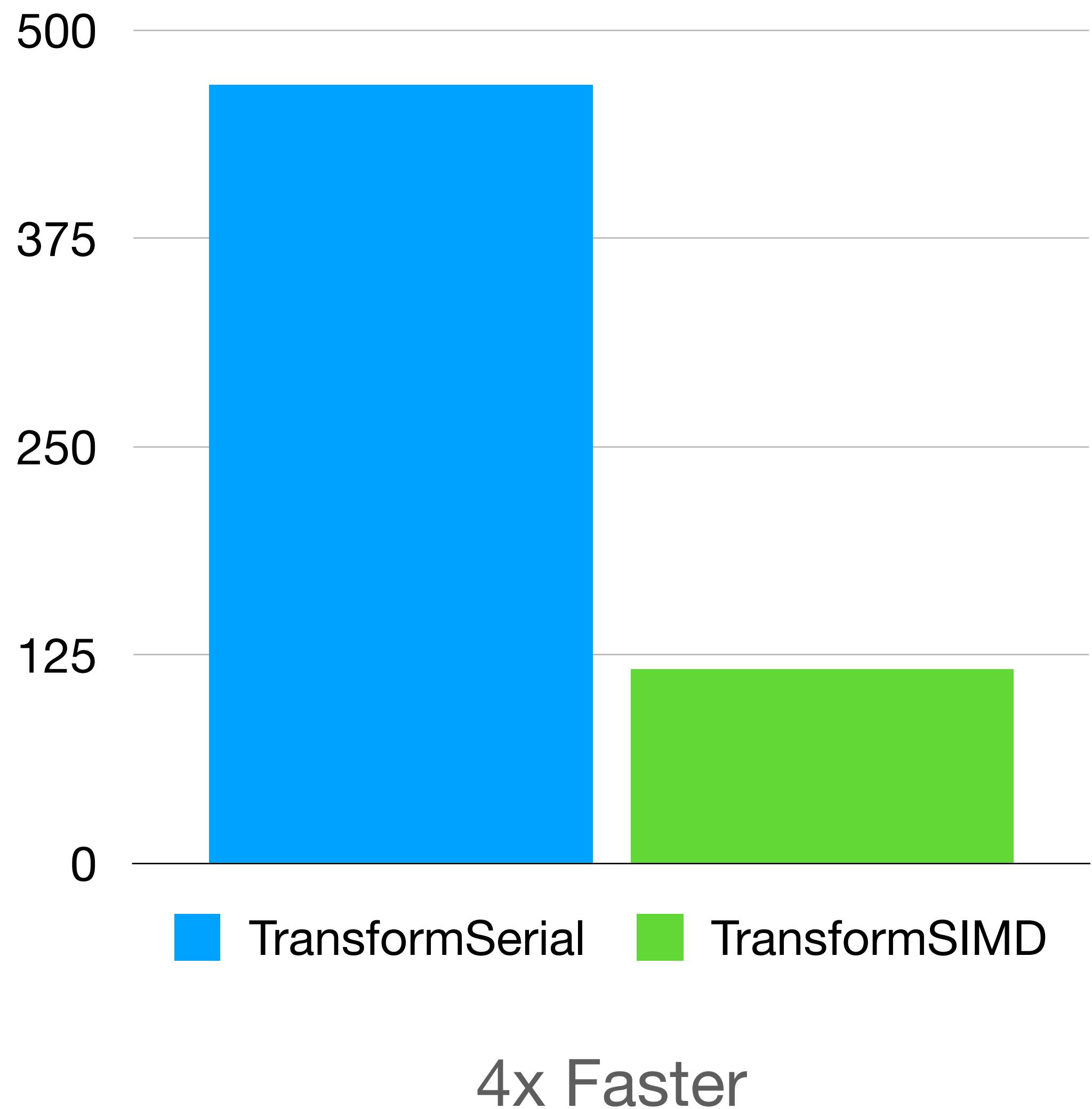
```

// Return 256 random floats +-1.0f
static const std::vector<float>& getBufferData();

static void TransformXXX(benchmark::State& state)
{
    const auto& v = getBufferData();
    auto copy = v;

    for (auto _ : state)
    {
        std::transform (v.begin(), v.end(), copy.begin(), copy.end(),
                      [] (auto v1, auto v2) { return v1 + v2; });
    }
}
BENCHMARK(TransformXXX);

```

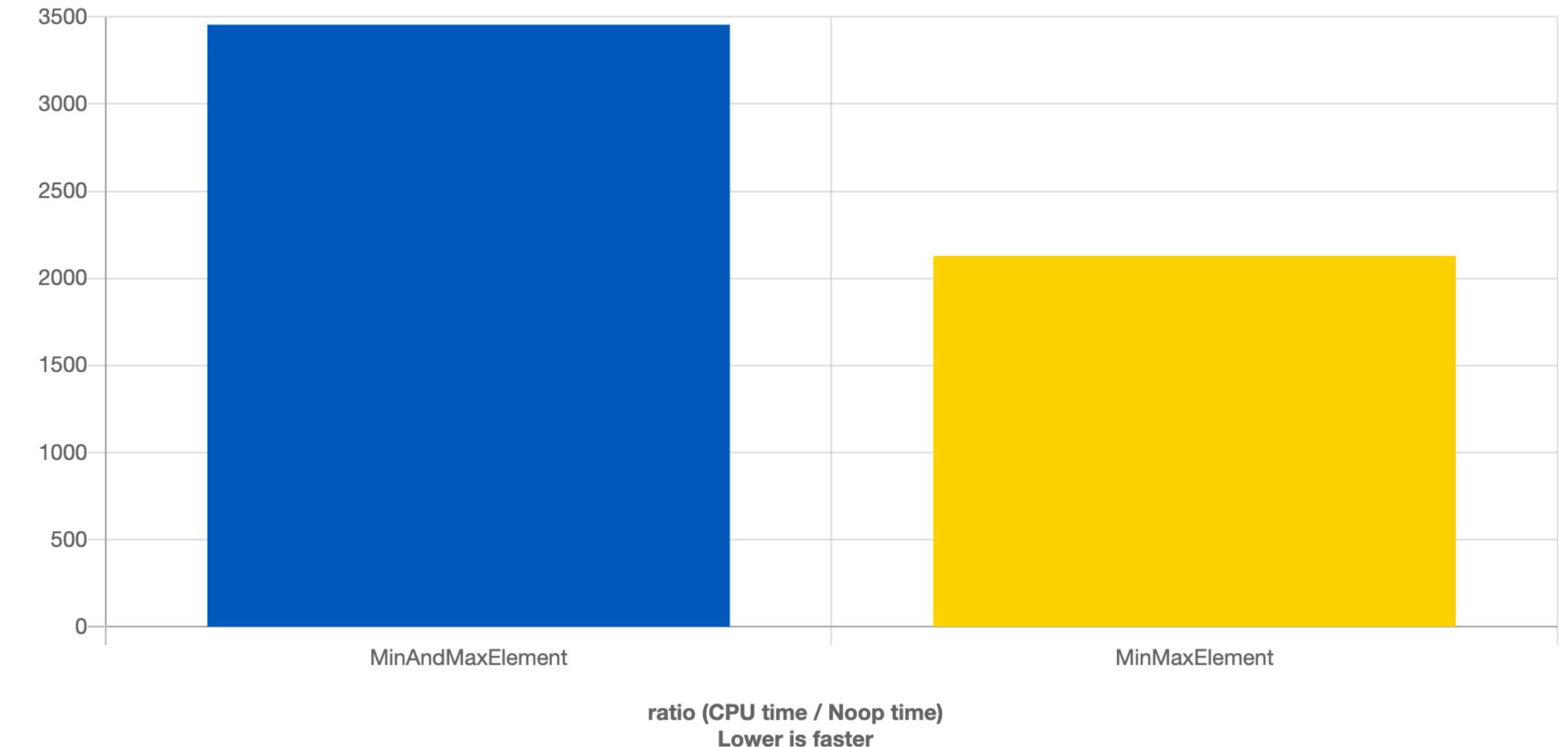


```
#include <algorithm>
```

```
const auto v = { 566, ... };
```

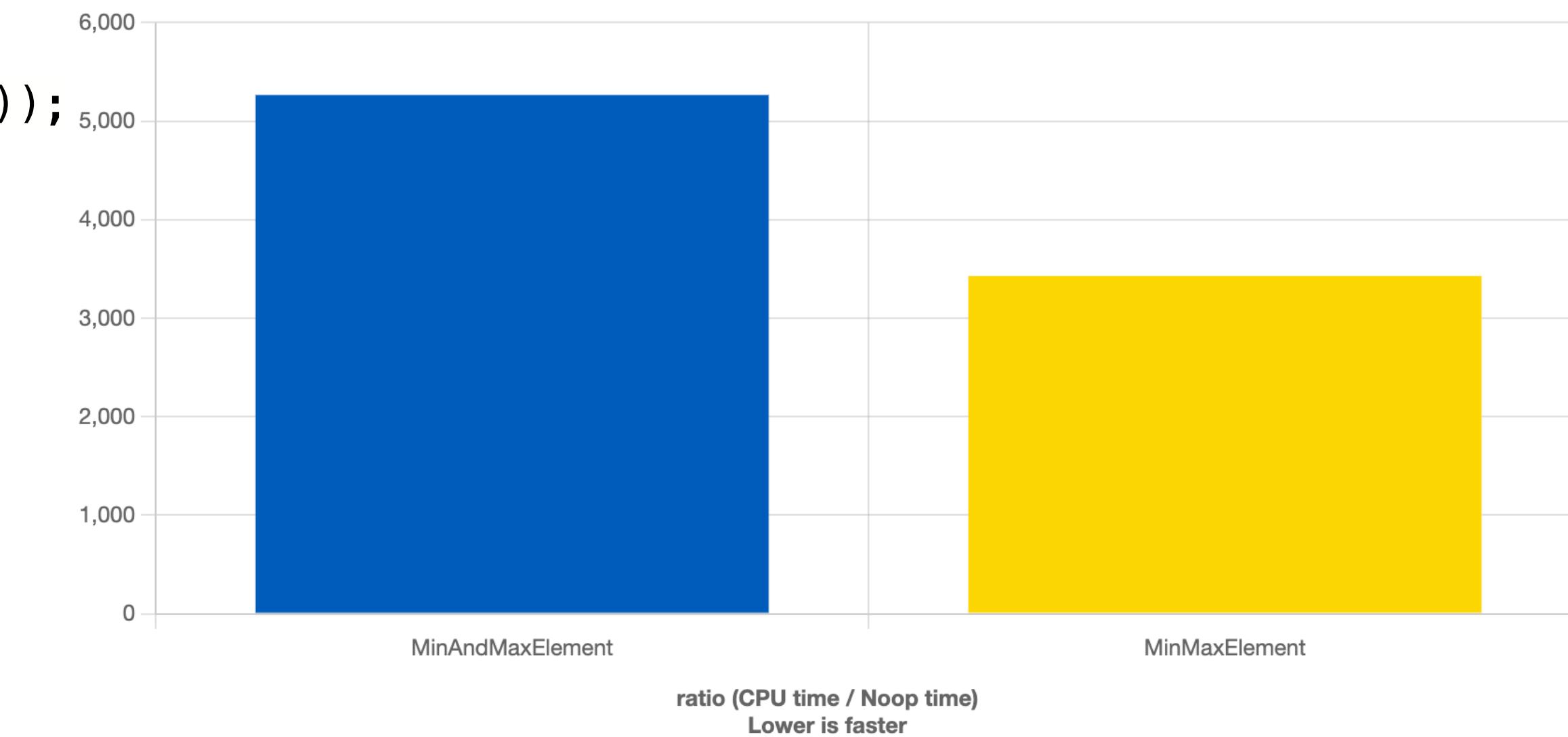
```
const auto min = std::min_element (begin(v), end(v));  
const auto max = std::max_element (begin(v), end(v));
```

GCC 12.2/libstdc++



```
const auto [min, max] = std::minmax_element (begin(v), end(v));
```

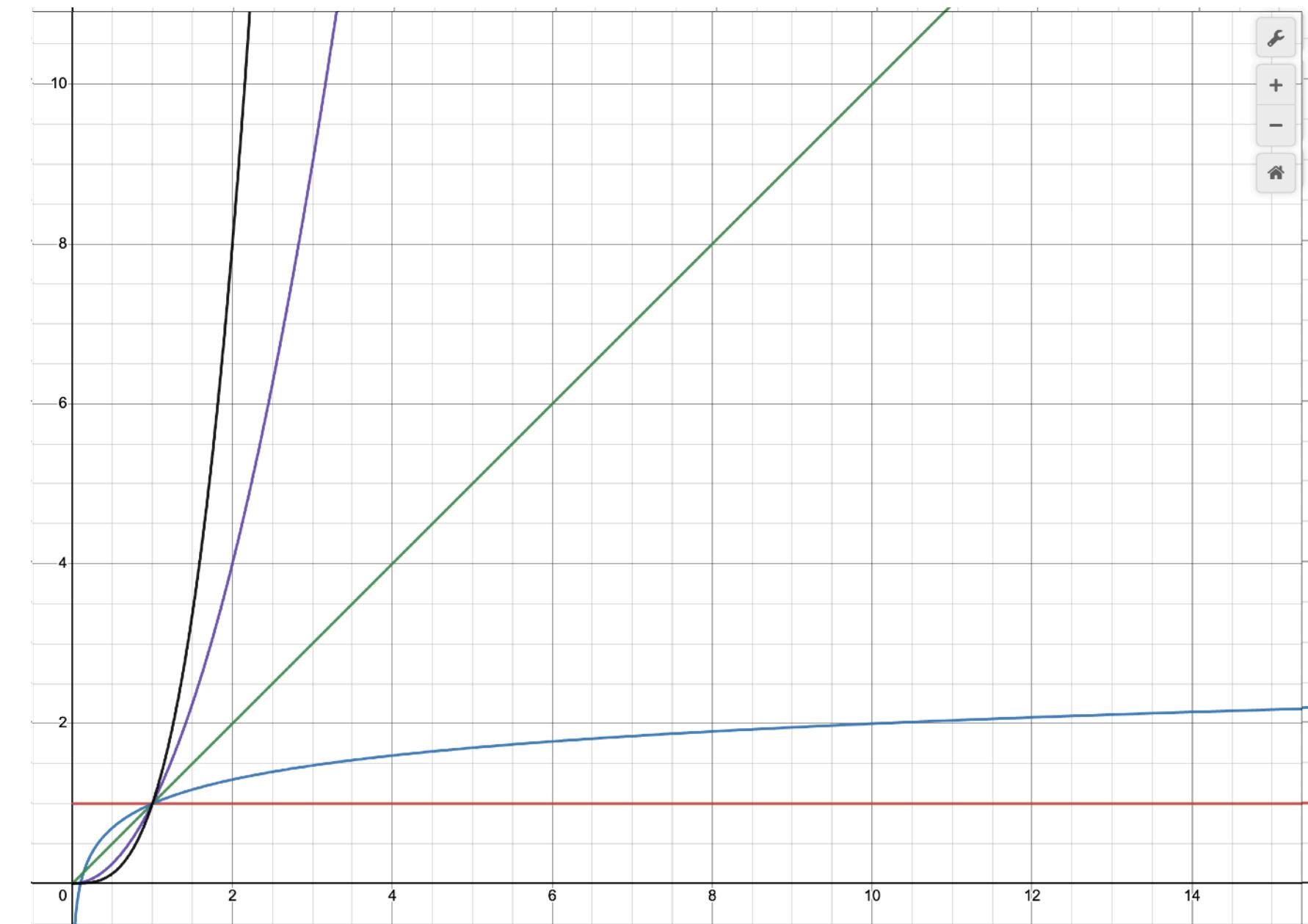
Clang 57/libstdc++



Techniques for Optimisation

3. Reducing algorithmic complexity

- Identify hotspots from profilers
- Rework algorithms to reduce complexity
- Big O notation
 - $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$ etc.
- Ensure your algorithm works the same way
 - Including contracts!
 - Testing saves you!



Example:
Edit::createNewItemID

```

EditItemID Edit::createNewItemID (const std::vector<EditItemID>& idsToAvoid) const
{
    // TODO: This *may* be slow under heavy load - keep an eye open for this
    // in case a smarter caching system is needed
    auto existingIDs = EditItemID::findAllIDs (state);

    existingIDs.insert (existingIDs.end(), idsToAvoid.begin(), idsToAvoid.end());
    existingIDs.insert (existingIDs.end(), usedIDs.begin(), usedIDs.end());
    trackCache.visitItems ([&] (auto i) { existingIDs.push_back (i->itemID); });
    clipCache.visitItems ([&] (auto i) { existingIDs.push_back (i->itemID); });

    std::sort (existingIDs.begin(), existingIDs.end());
    auto newID = EditItemID::findFirstIDNotIn (existingIDs);
    jassert (usedIDs.find (newID) == usedIDs.end());
    usedIDs.insert (newID);

    return newID;
}

```

```

std::vector<EditItemID> EditItemID::findAllIDs (const juce::ValueTree& v)
{
    std::vector<EditItemID> ids;

    IDRemapping::visitAllIDDecls (v, [&] (const juce::var& oldID)
    {
        auto i = EditItemID::fromVar (oldID);

        if (i.isValid())
            ids.push_back (i);
    });

    return ids;
}

```

```

template <typename Visitor>
static void visitAllIDDecls (const juce::ValueTree& v, Visitor&& visitor)
{
    for (int i = 0; i < v.getNumProperties(); ++i)
    {
        auto propName = v.getPropertyName (i);

        if (isIDDeclaration (propName))
            visitor (v.getProperty (propName));
    }

    for (const auto& child : v)
        visitAllIDDecls (child, visitor);
}

```

```

EditItemID Edit::createNewItemID (const std::vector<EditItemID>& idsToAvoid) const
{
    if (nextID == 0)
    {
        auto existingIDs = EditItemID::findAllIDs (state);

        existingIDs.insert (existingIDs.end(), idsToAvoid.begin(), idsToAvoid.end());

        trackCache.visitItems ([&] (auto i) { existingIDs.push_back (i->itemID); });
        clipCache.visitItems ([&] (auto i) { existingIDs.push_back (i->itemID); });

        std::sort (existingIDs.begin(), existingIDs.end());
        nextID = existingIDs.empty() ? 1001 : (existingIDs.back().getRawID() + 1);

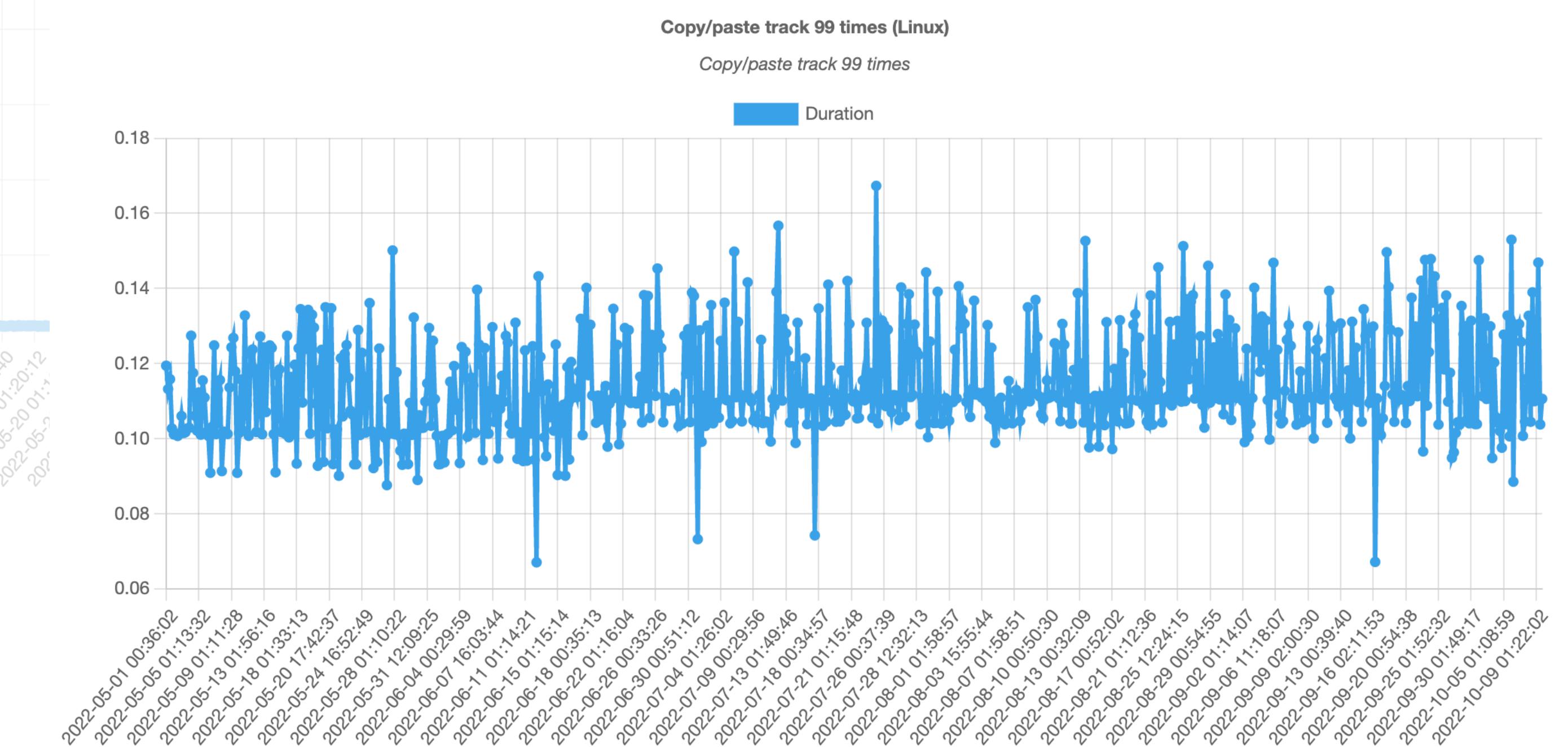
        #if JUCE_DEBUG
        usedIDs.insert (existingIDs.begin(), existingIDs.end());
        #endif
    }

    auto newID = EditItemID::fromRawID (nextID++);

    #if JUCE_DEBUG
    jassert (usedIDs.find (newID) == usedIDs.end());
    usedIDs.insert (newID);
    #endif

    return newID;
}

```



100x

- Reduced complexity from $O(2n)$ to $O(1)$
- Behaviour has changed
 - ID now always increases
 - Could wrap if called `std::numeric_limits<uint64_t>::max()` times
 - 18446744073709551615 (18×10^{18})
 - That's quite a lot of tracks/clips/plugins
- Contract has stayed the same - returns a unique ID
 - Checked with an assertion

Techniques for Optimisation

4. Caching appropriate data

- If data is accessed and manipulated in the same way frequently, it may be faster to store the manipulated data somewhere
- Difficulties involve:
 - Keeping track of the cache/dirty state
 - Knowing when to clean up the cache
- Uses additional memory

Example:
MidiClip::getSequenceLooped()

```
class MidiClip : public Clip
{
private:
    mutable std::unique_ptr<MidiList> cachedLoopedSequence;
```

```
MidiList& MidiClip::getSequenceLooped() const
{
    if (!isLooping())
        return getSequence();

    if (cachedLoopedSequence == nullptr)
        cachedLoopedSequence = createSequenceLooped(getSequence());
}

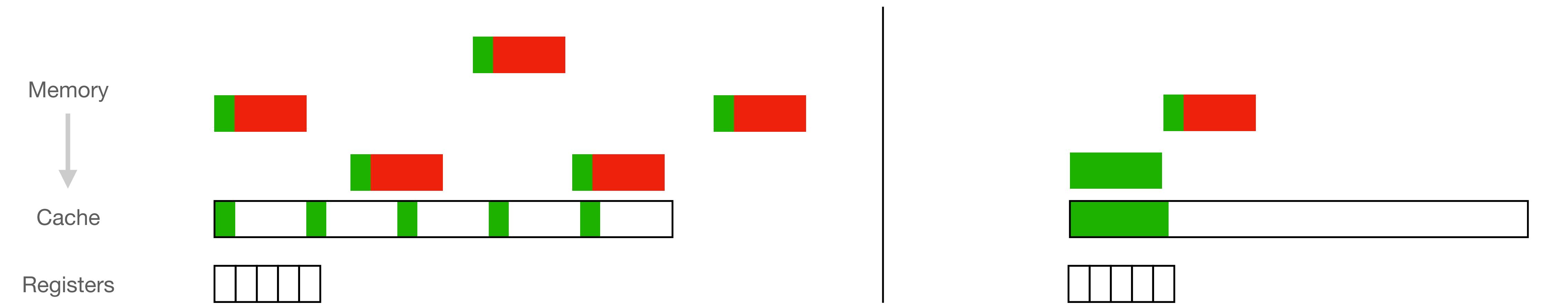
return *cachedLoopedSequence;
}
```

Techniques for Optimisation

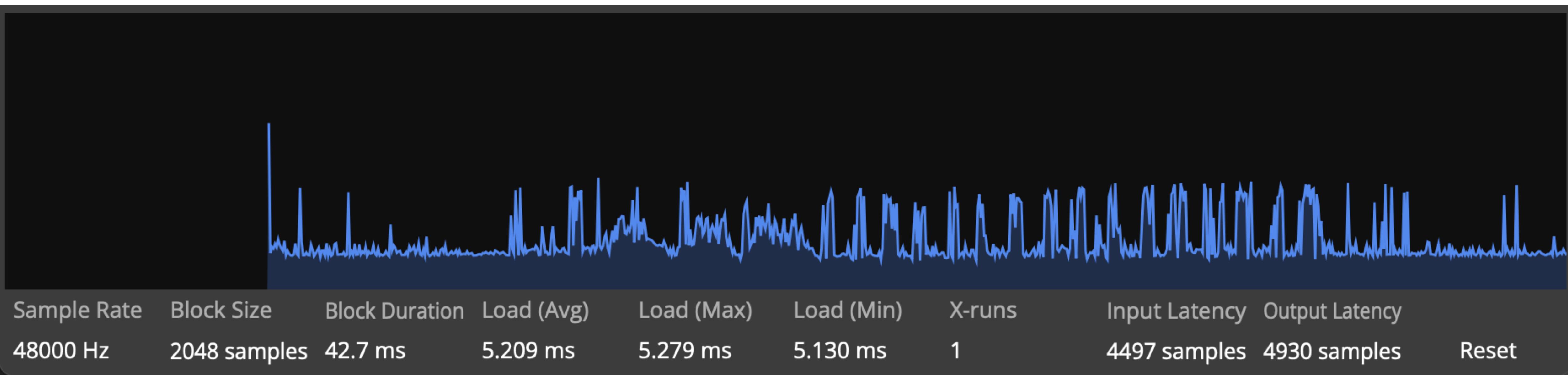
5. Reducing memory footprint

- Cache misses stall the CPU effectively wasting cycles
- Reducing the amount of memory a performance sensitive section of code uses can reduce the time to execute
 - Memory is more likely to be in lower cache levels
 - Lower cache levels are MUCH faster
- If possible, use contiguous memory

- Memory is more likely to be in lower cache levels
- Lower cache levels are MUCH faster



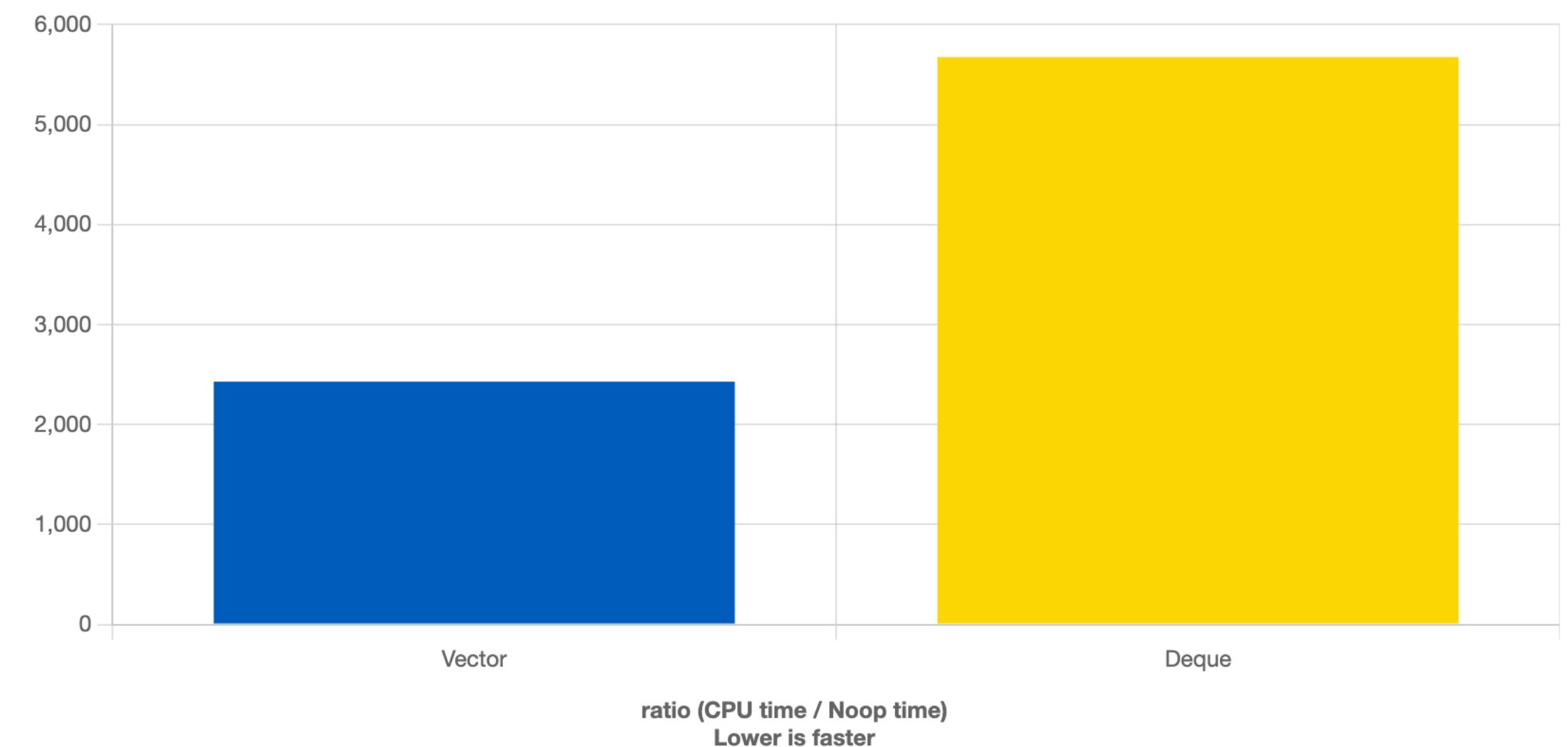
Example: CPU Usage Graph



```
numElements = 1,000
```

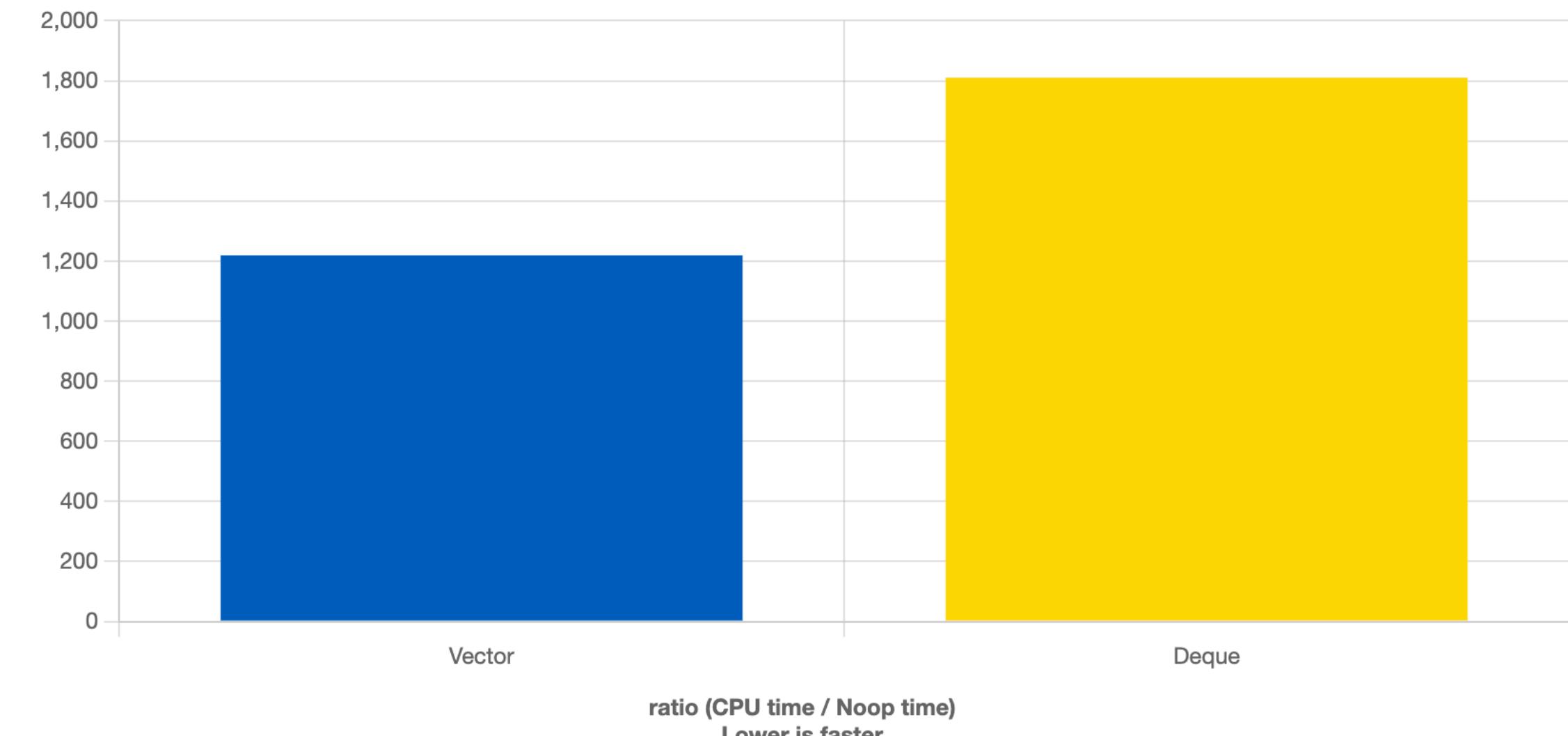
CGG 13.2 - libstdc++

```
30     std::vector<float> v;
31
32     for (auto _ : state)
33     {
34         v.insert (v.begin(), get_next_val());
35
36         if (v.size() > numElements)
37             v.pop_back();
38
39         readData (v);
40     }
41 }
```

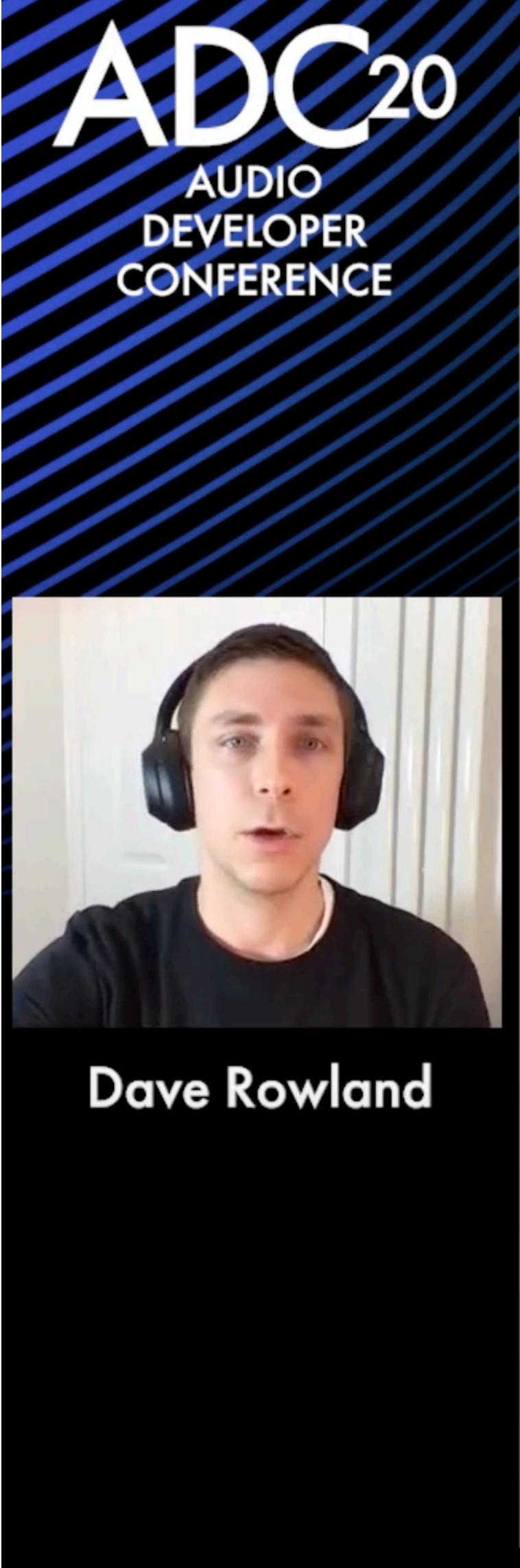


```
48     std::deque<float> v;
49
50     for (auto _ : state)
51     {
52         v.push_front (get_next_val());
53
54         if (v.size() > numElements)
55             v.pop_back();
56
57         readData (v);
58     }
59 }
```

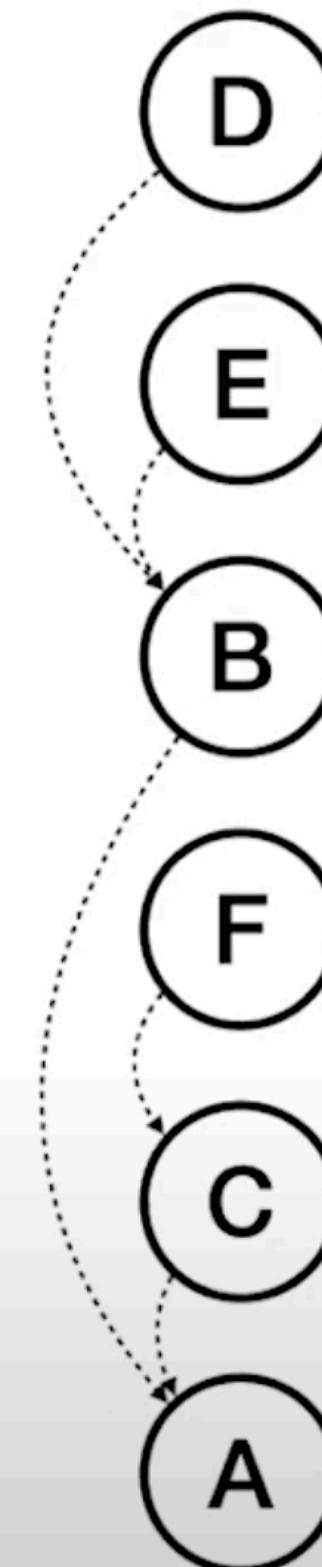
Clang 17 - libc++



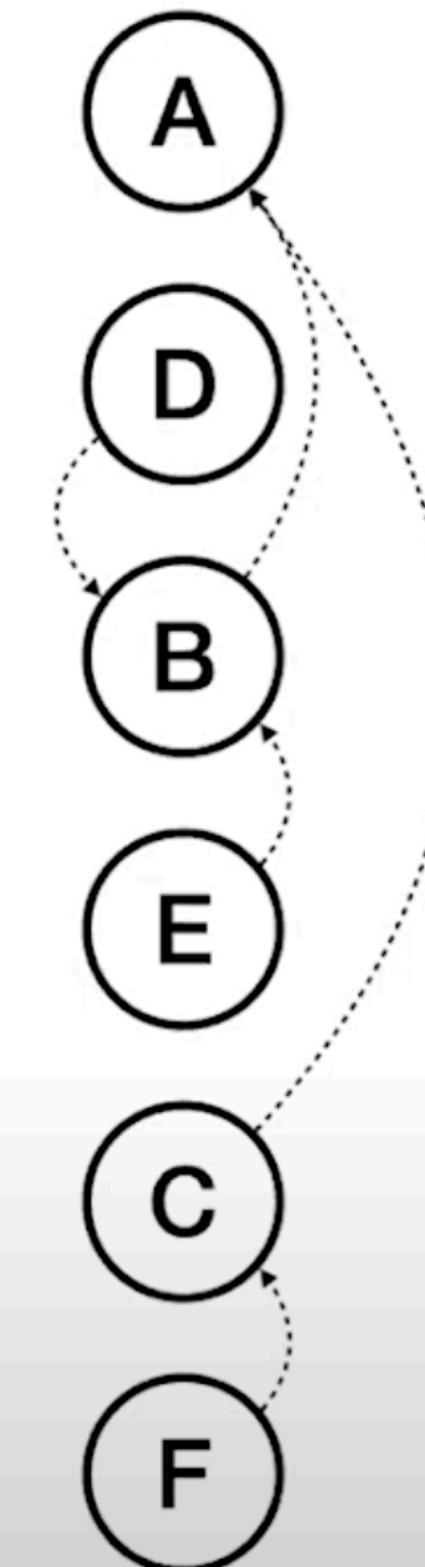
4. Multi-threading, CPUs and memory

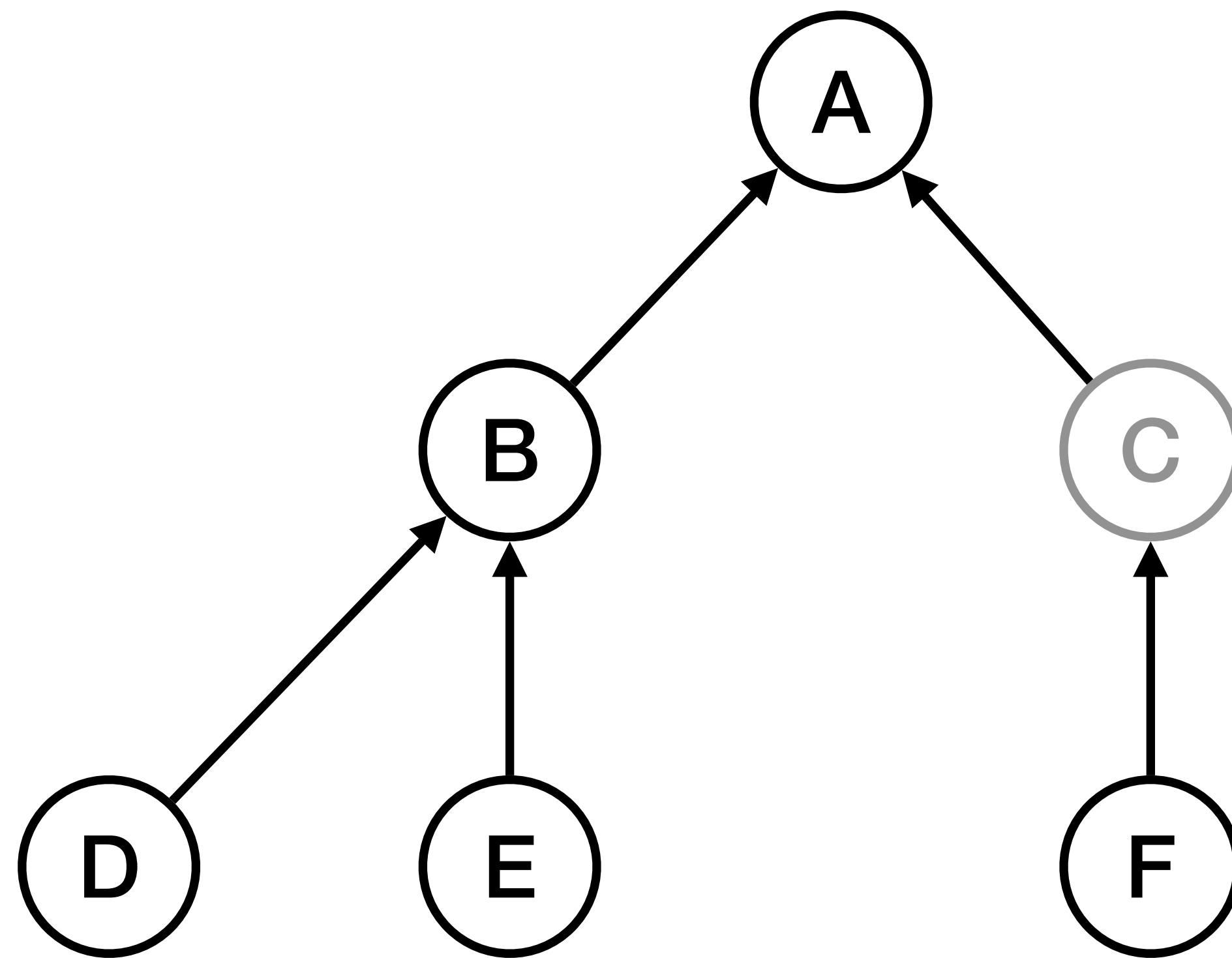


Post-ordered DFS:



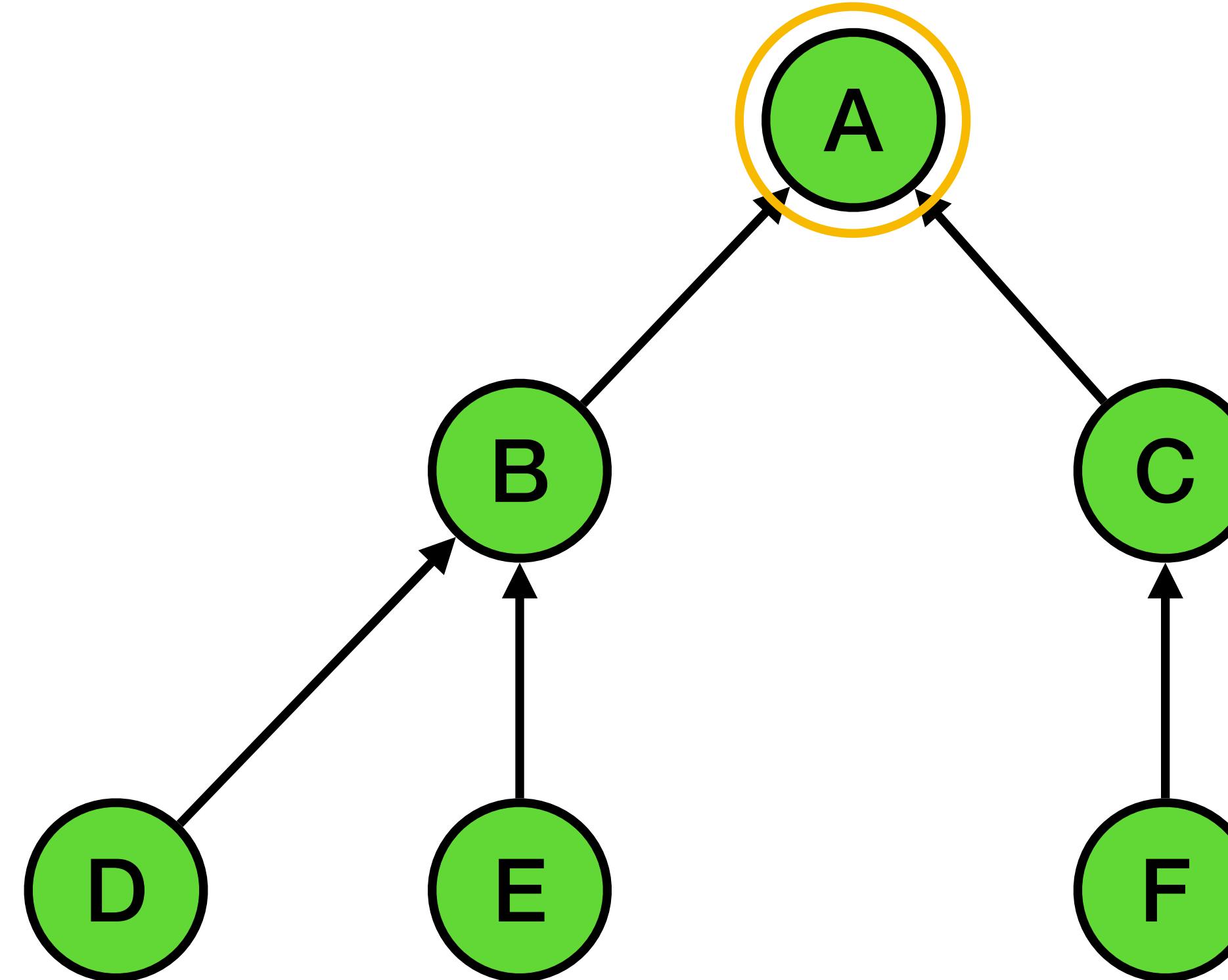
Pre-ordered DFS:

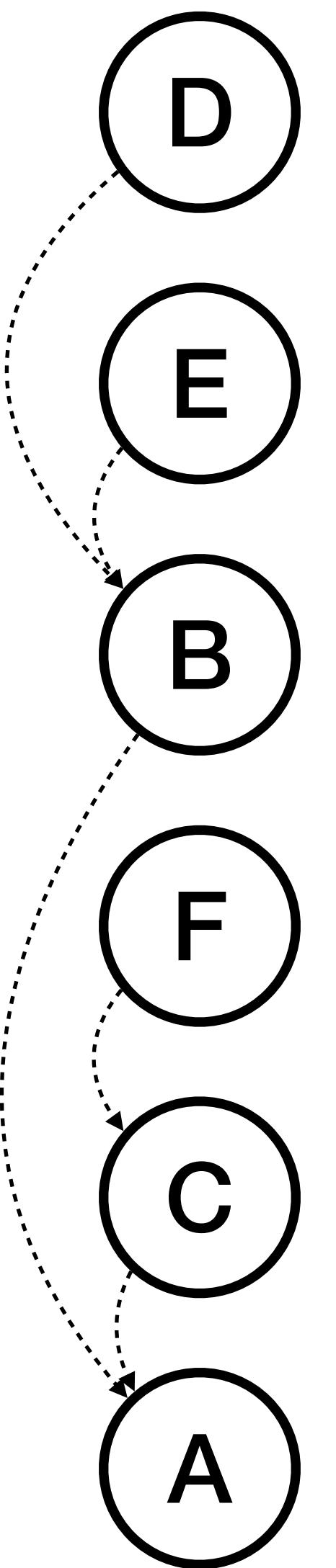




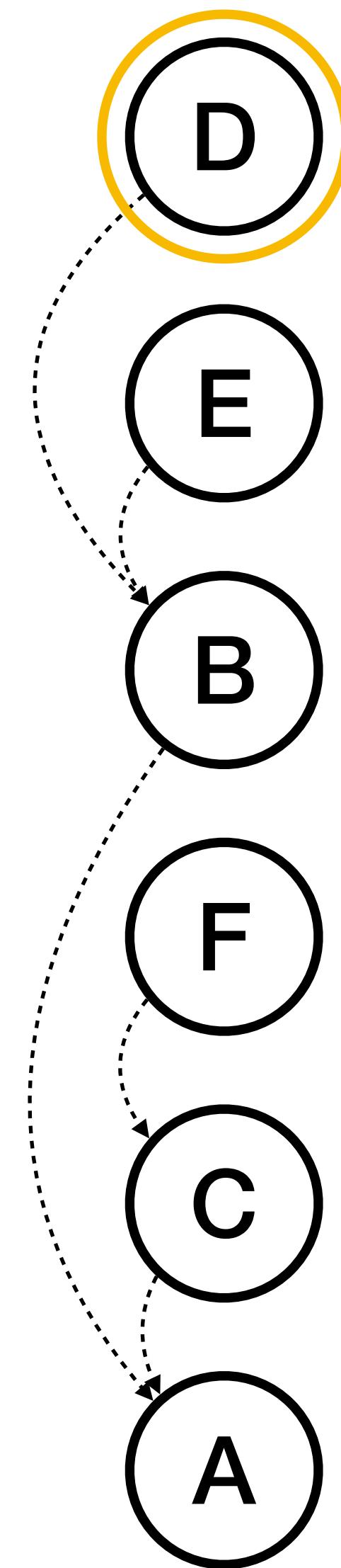
Directed Acyclic Graph

- Post-ordered DFS:

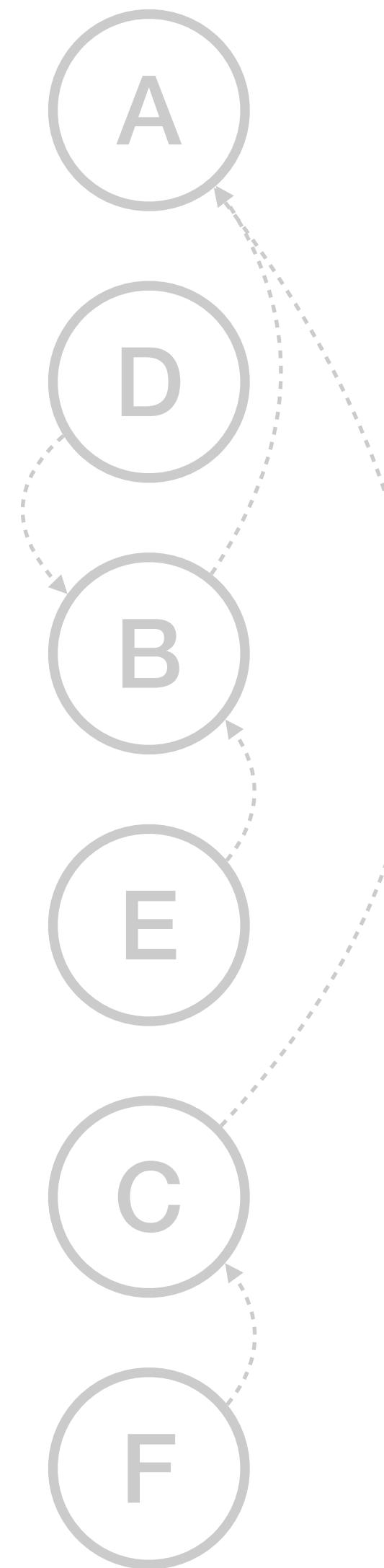




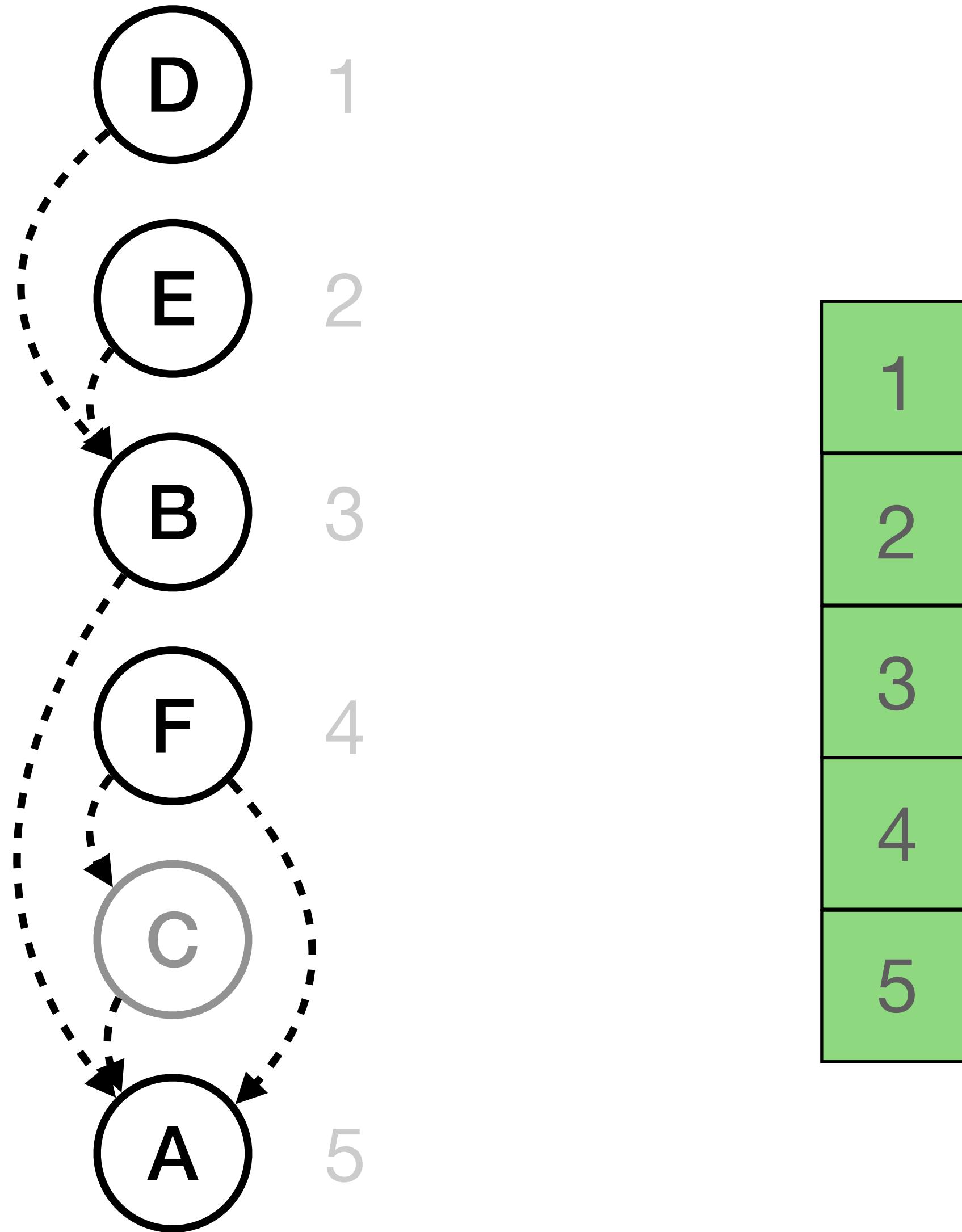
Post-ordered DFS: ✓



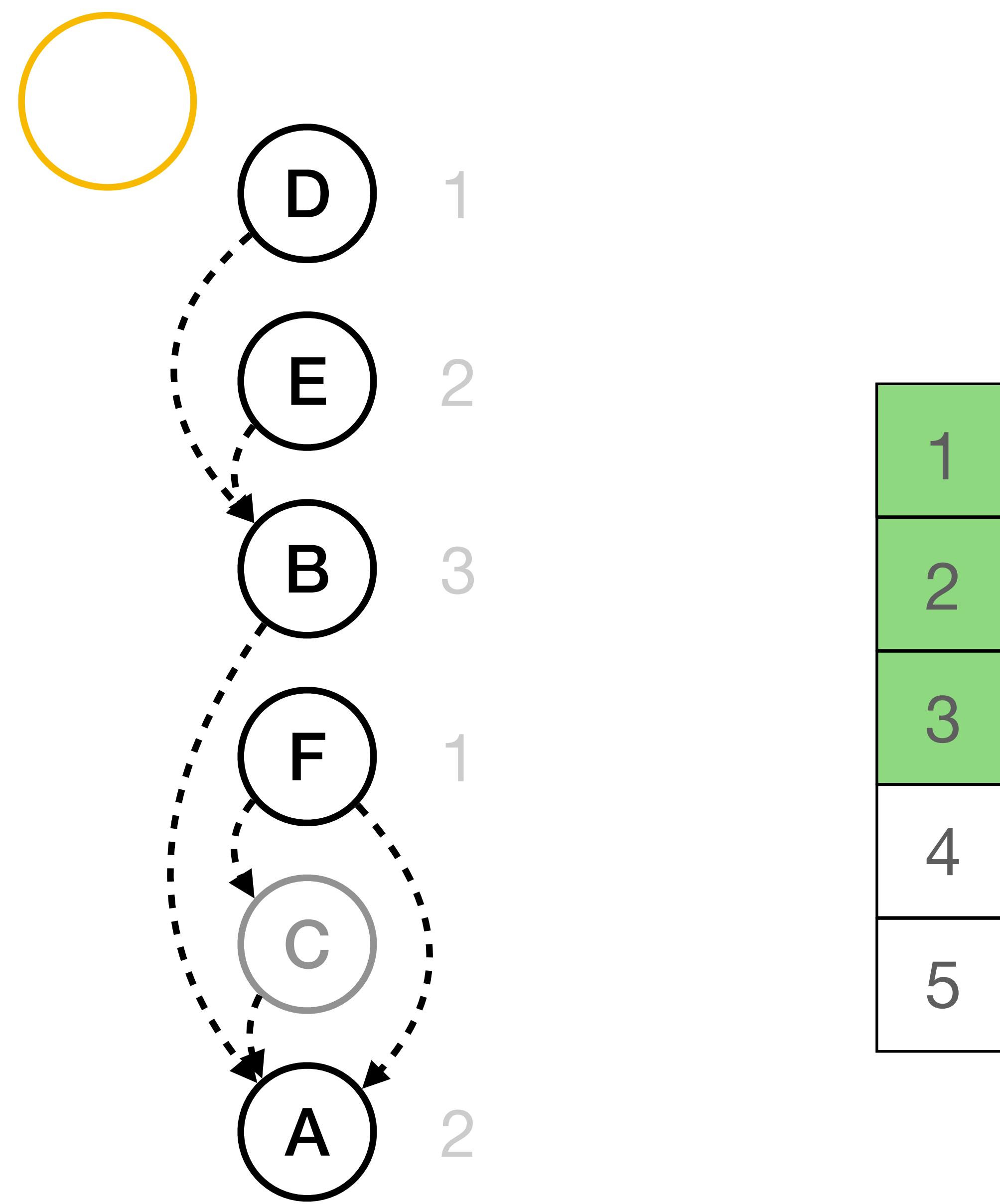
Pre-ordered DFS: ✗



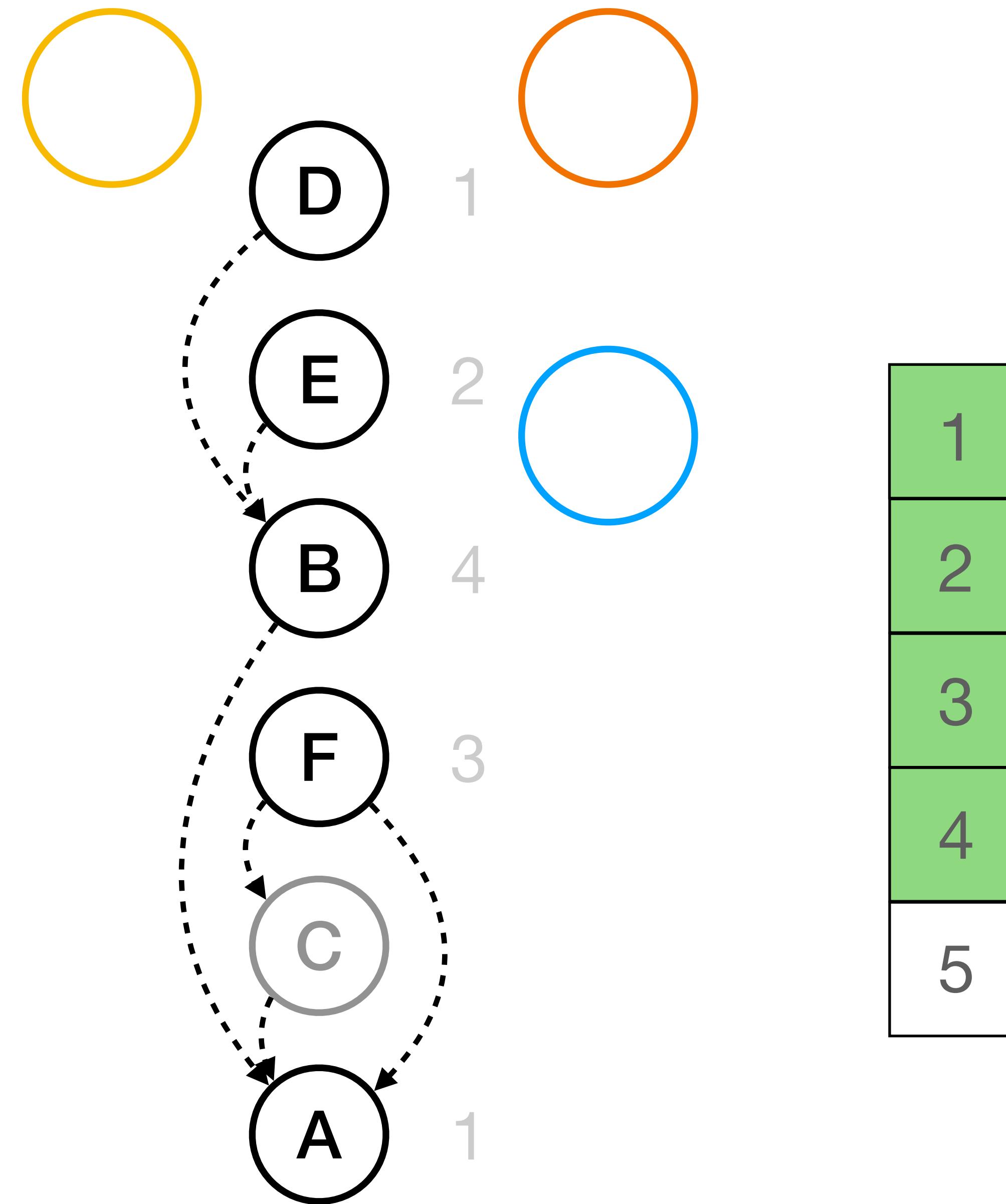
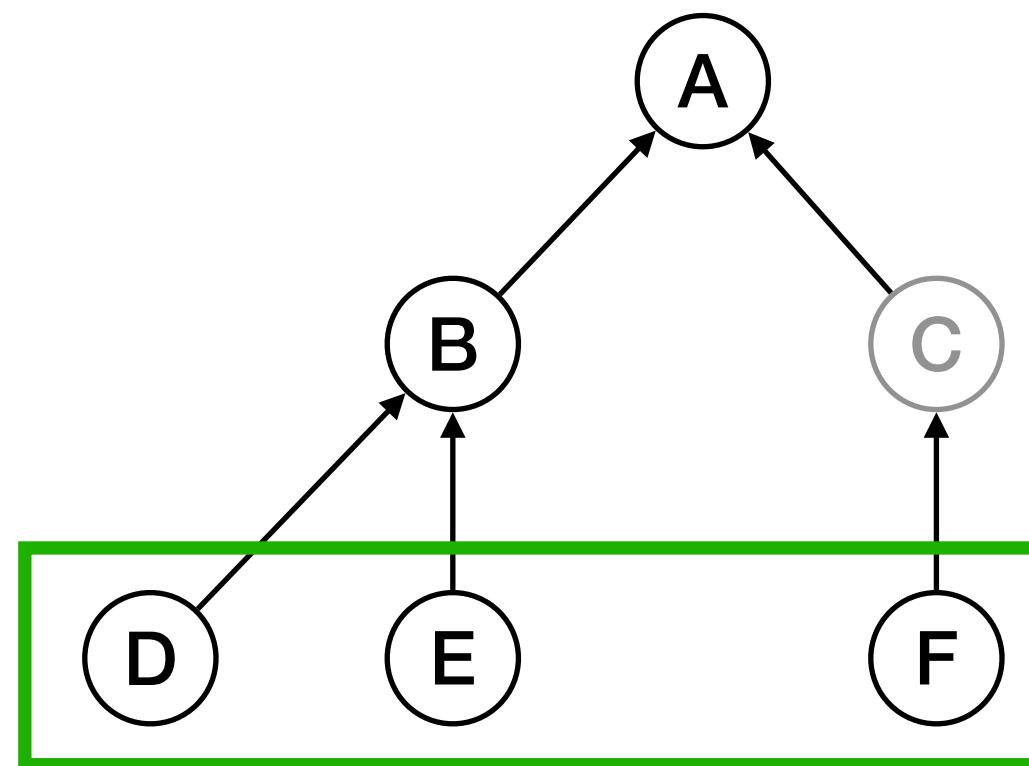
Per-node Assignment



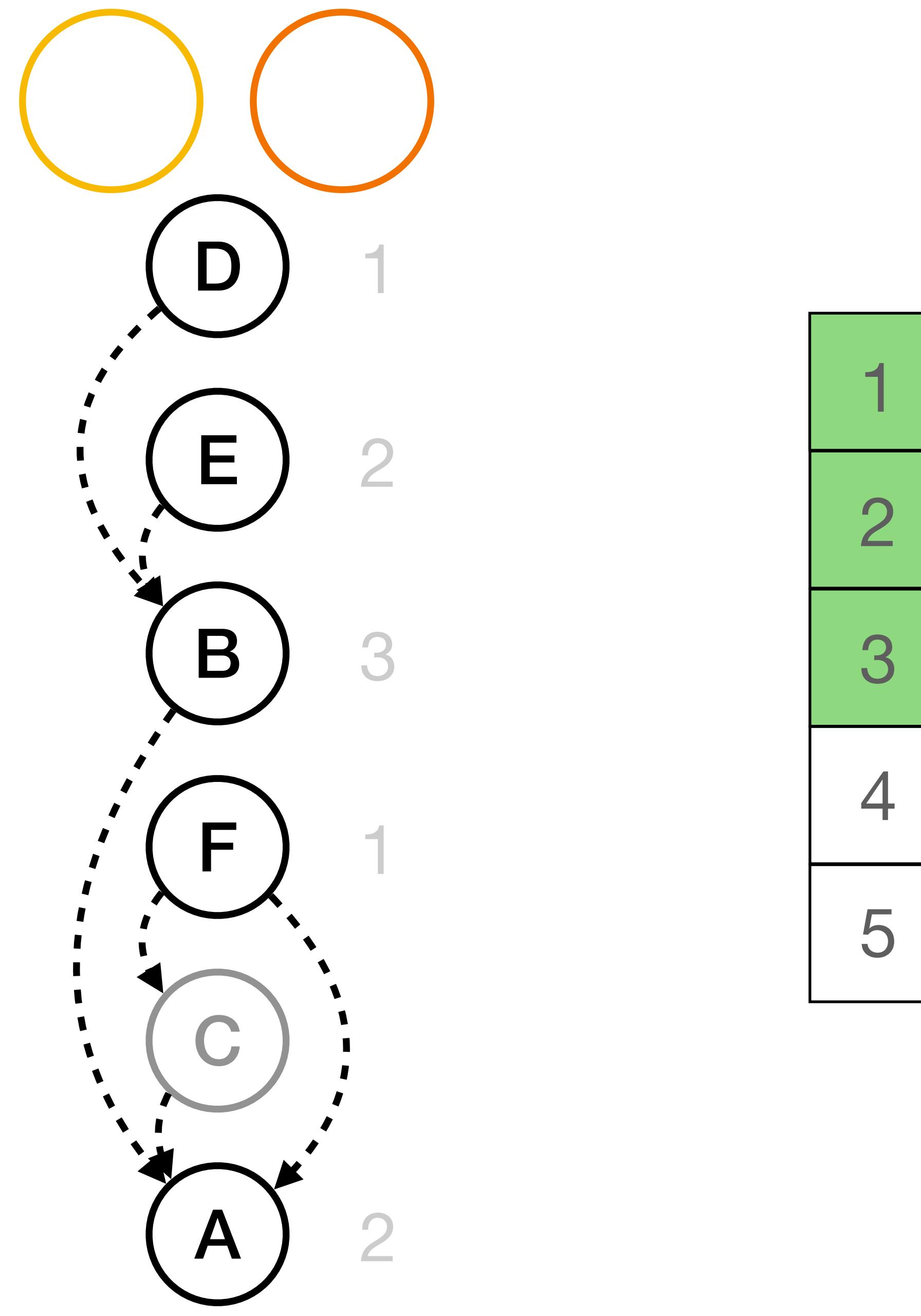
Single Threaded Analysis



Multi Threaded Pre-process Analysis

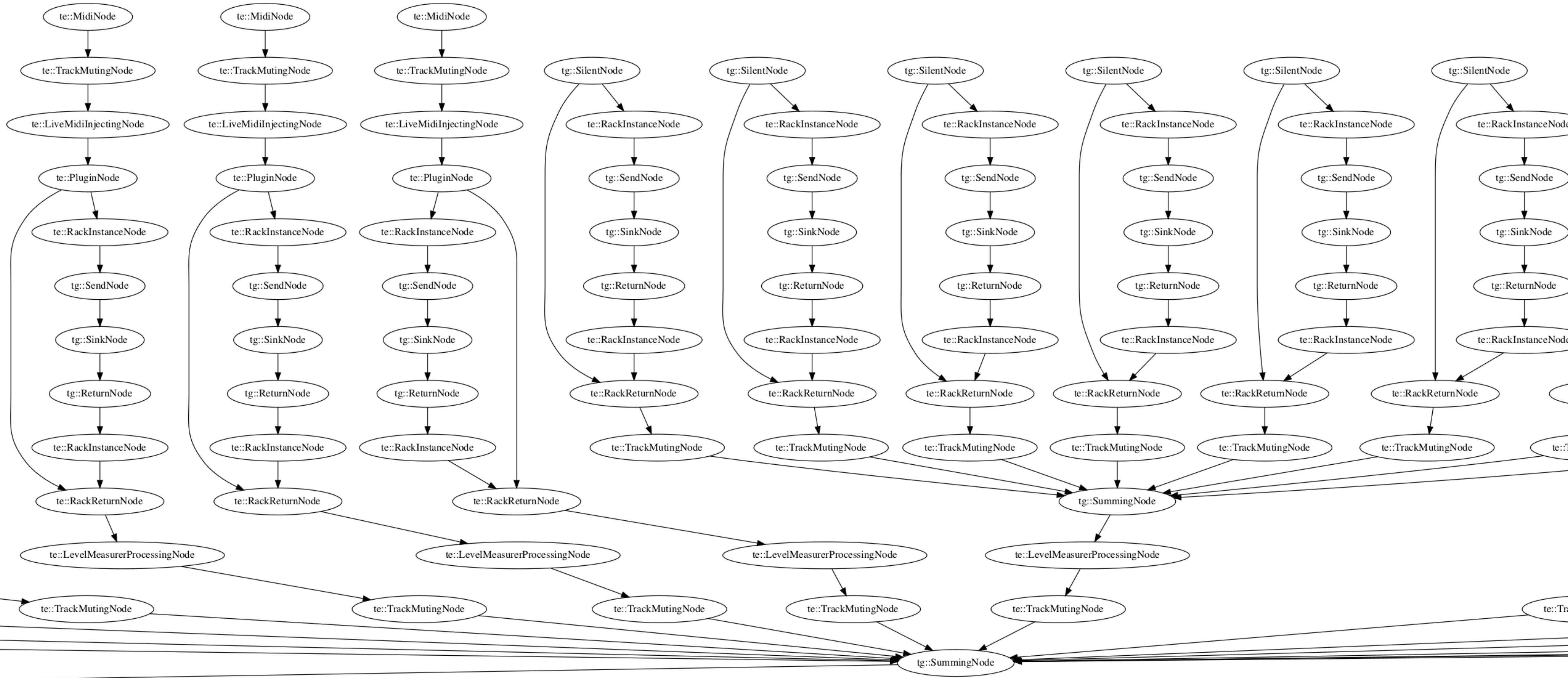


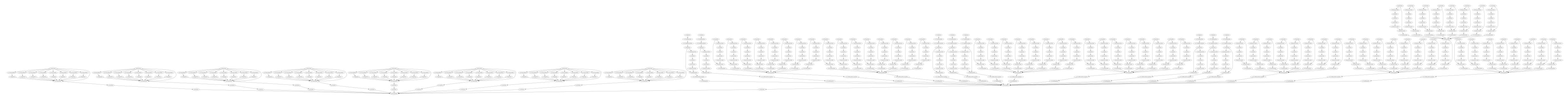
Multi Threaded Run-time Memory Assignment



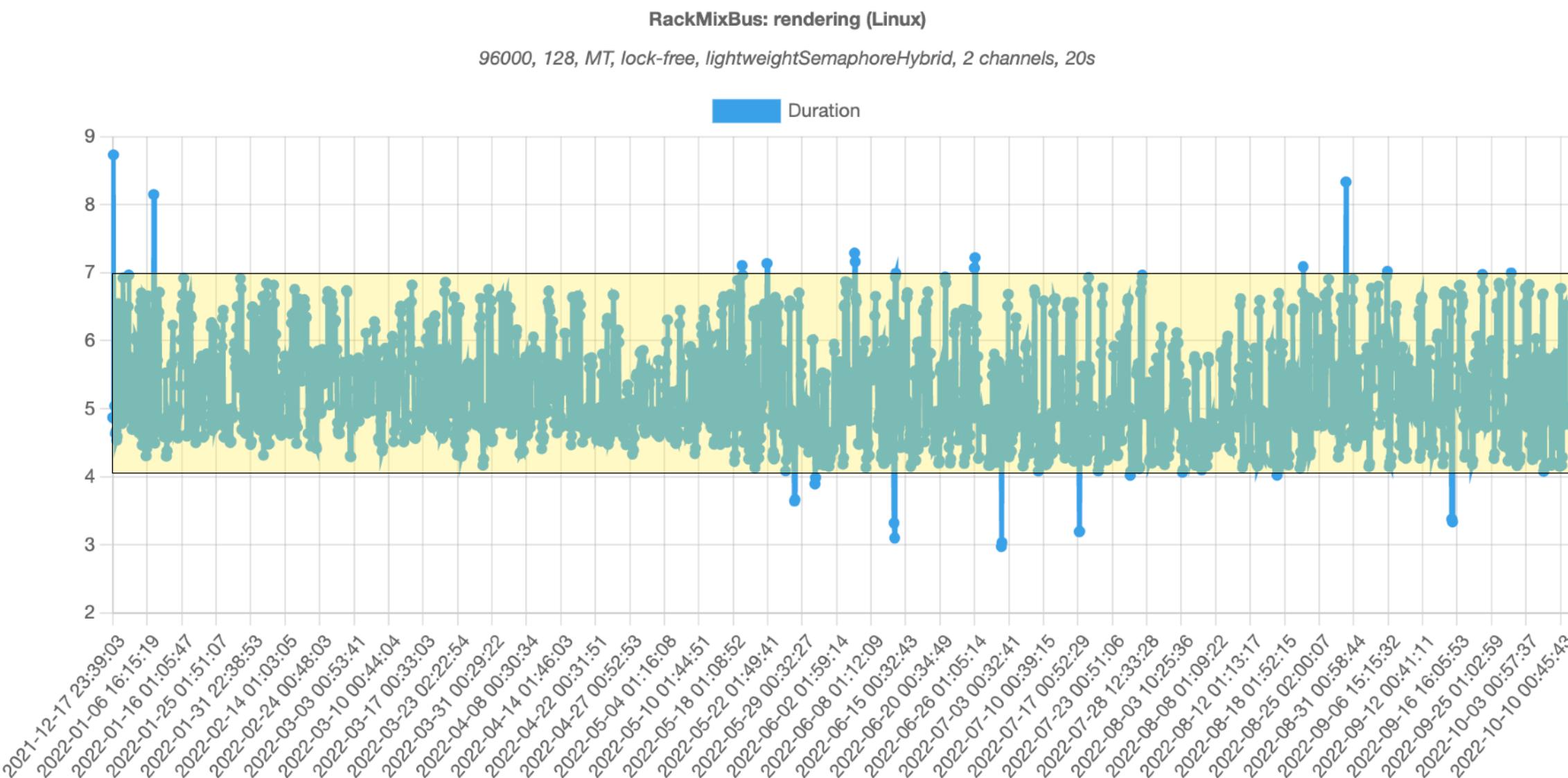
Multi-threaded Assignment

- Difficult to reason about
- Varies a lot depending on:
 - Graph size
 - Graph structure
 - Node memory size
 - Number of threads
- **Need to measure/benchmark!**

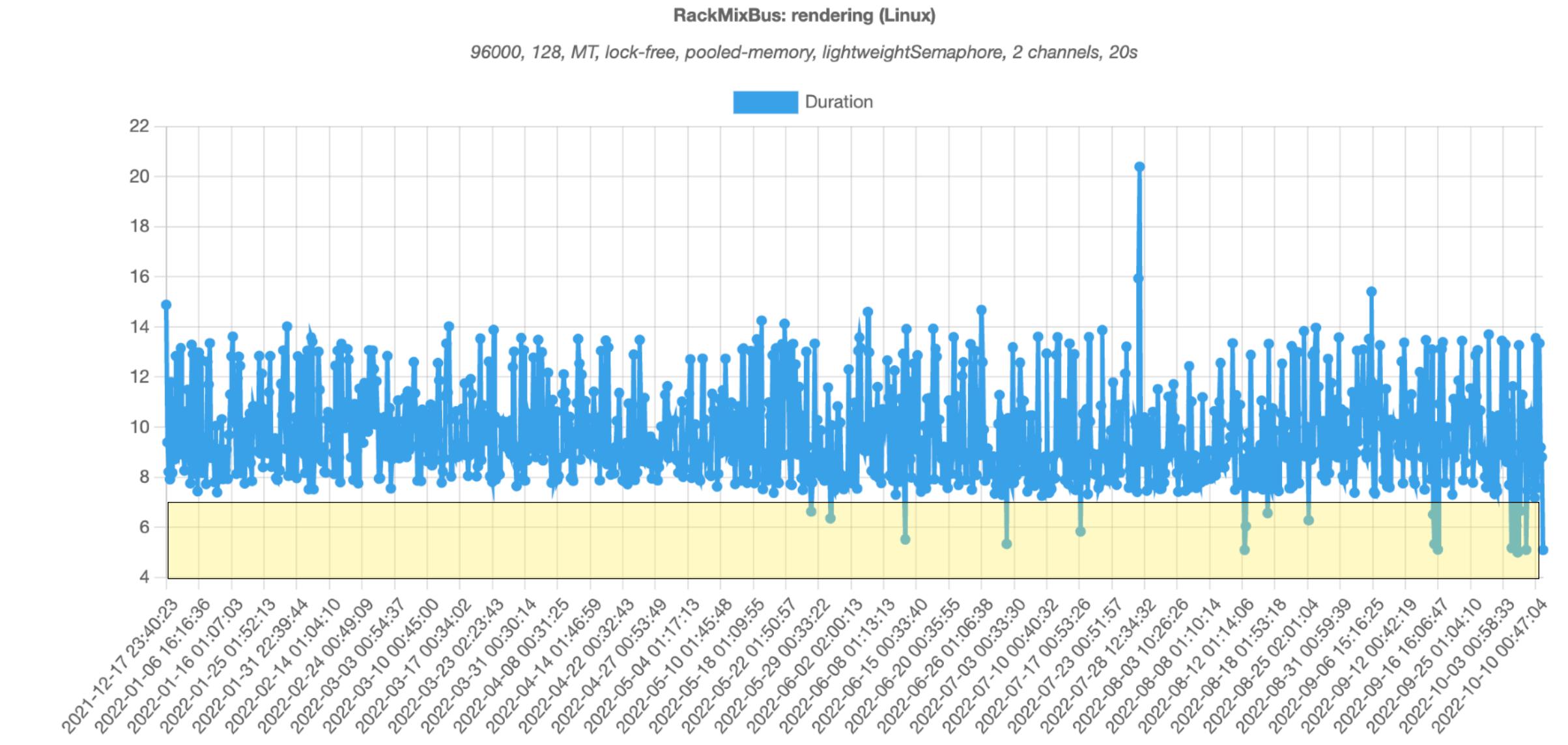




Each Node Has its Own Memory

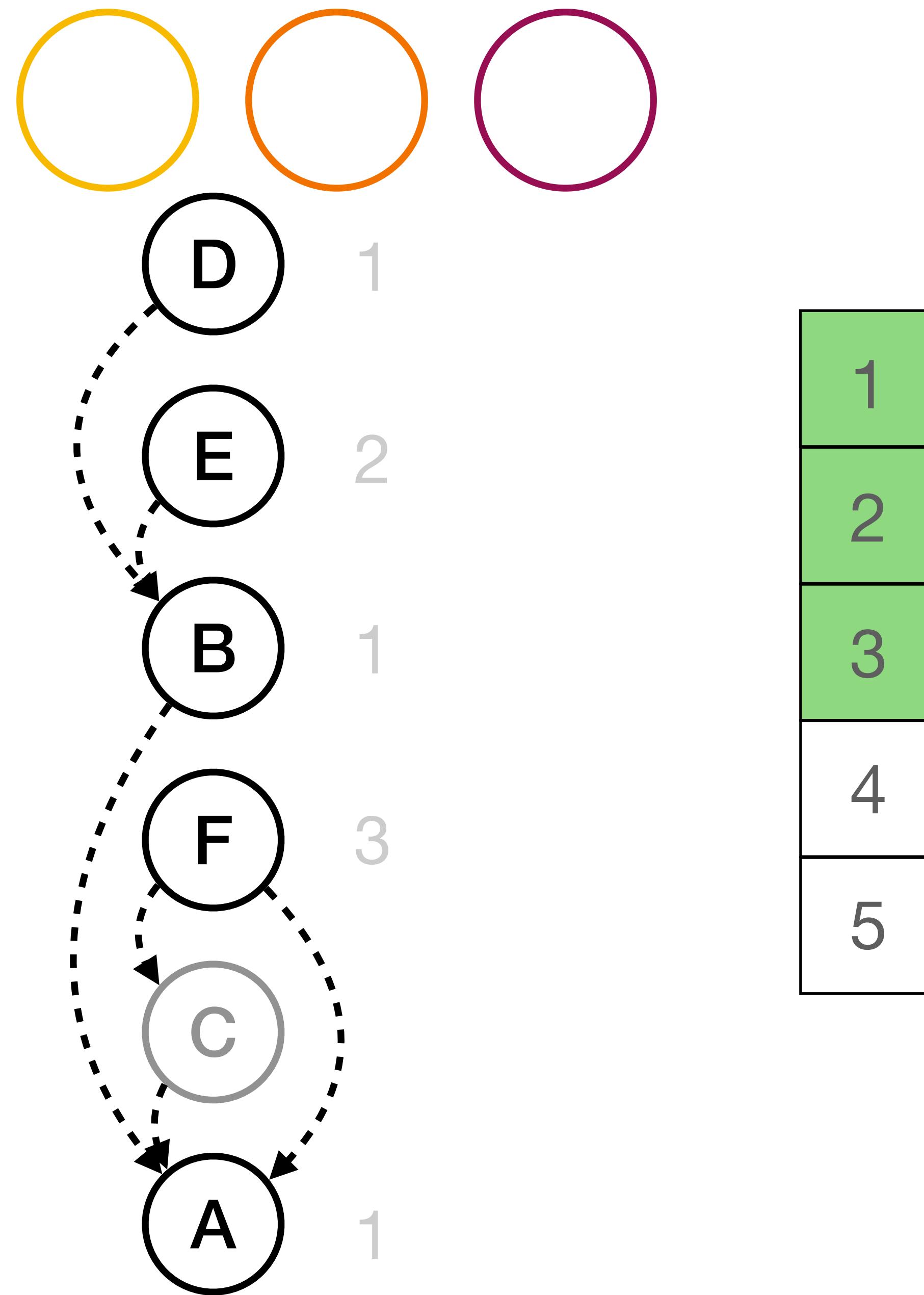


Run-time Memory Assignment

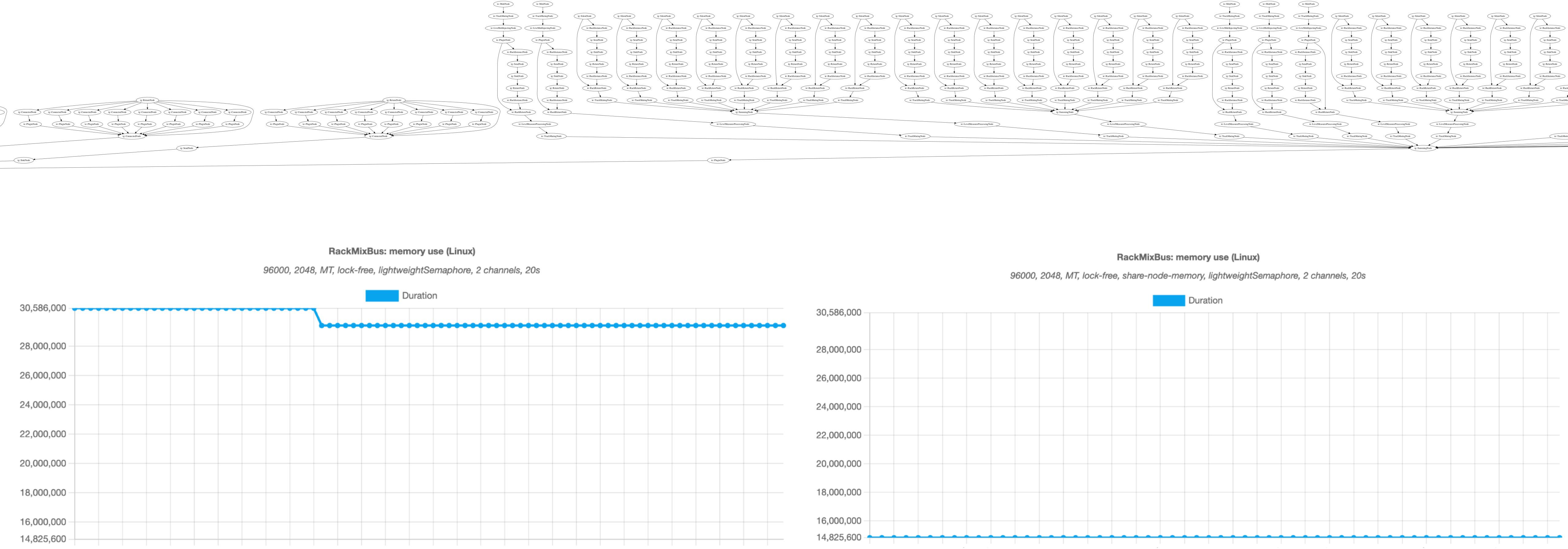


- The overhead of using less memory outweighed the cache benefits
 - Complexity in run-time assignment of memory
 - Might be different on more constrained platforms
 - Or different graph configurations
 - E.g. more nodes, larger channel counts, larger buffer sizes etc.
 - May need some heuristics of when to adapt

Race-safe Static Memory Assignment



- Pros:
 - Same memory usage as optimal single threaded pattern*
 - Allows other optimisations such as avoiding copies
- Cons:
 - Has to be implemented per-Node
 - Additional complexity to find single-output Nodes





7% Improvement

Lessons Learned from Optimising a High Performance Library

David Rowland
 X @drowaudio

Slides/video:
github.com/drowaudio/presentations

tracktion code:
github.com/Tracktion/tracktion_engine