

+ 25

CTRACK

Lightweight C++ Performance Tracking for
Bottleneck Discovery

GRISCHA HAUSER

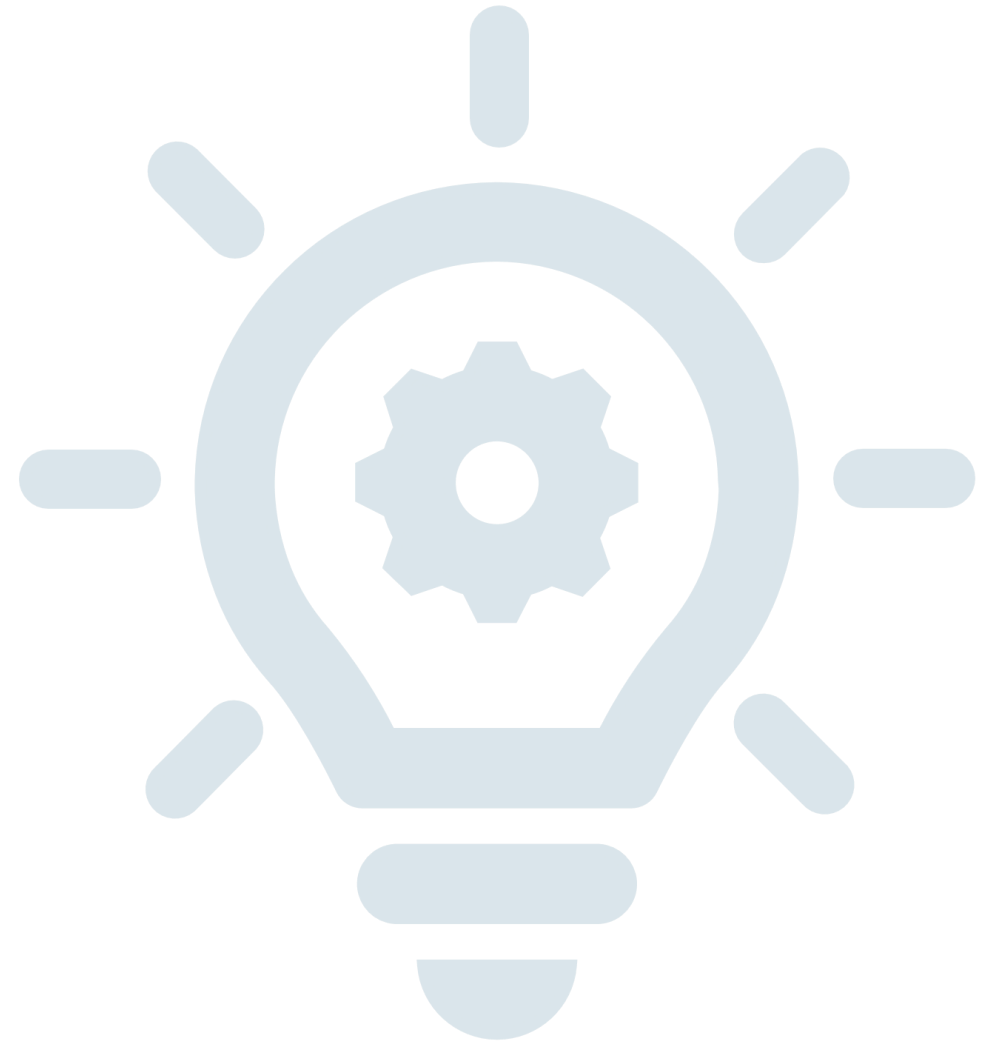


20
25



"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs"

Donald Knuth



Current Landscape

Benchmarking

- Google Benchmark
- Nanobench
- CppBenchmark
- ...

And many more!

Profilers



• CPU Profilers:

- perf
- Tracy Profiler
- VS Performance Profiler
- Very Sleepy



• Deep Hardware / GPU Profilers:

- Intel VTune Profiler
- AMD μ Prof
- NVIDIA Nsight

Sample-Based Profiling

- **What:** A low-overhead statistical analysis of a running program.
- **How:** Periodically "pauses" the application and records what function is running.
-  Overhead
-  **Accuracy**

Instrumentation-Based Profiling

- **What it is:** A highly-detailed trace of program execution.
- **How it works:** Injects measurement code (probes) at the entry and exit of functions.
-  **Accuracy**
-  **Overhead**

Micro-Benchmarking

- **What it is:** A controlled experiment to measure a single function or algorithm.
- **How it works:** Runs a function in a loop many times to get a stable average.

```
void important_function() {  
    auto start = high_resolution_clock::now();  
    // Simulate some work  
    this_thread::sleep_for(milliseconds(100));  
    auto end = high_resolution_clock::now();  
    auto duration = duration_cast<milliseconds>(end - start);  
    cout << "took: " << duration.count() << " ms" << endl;  
}
```

What we need in practice

Why this is hard

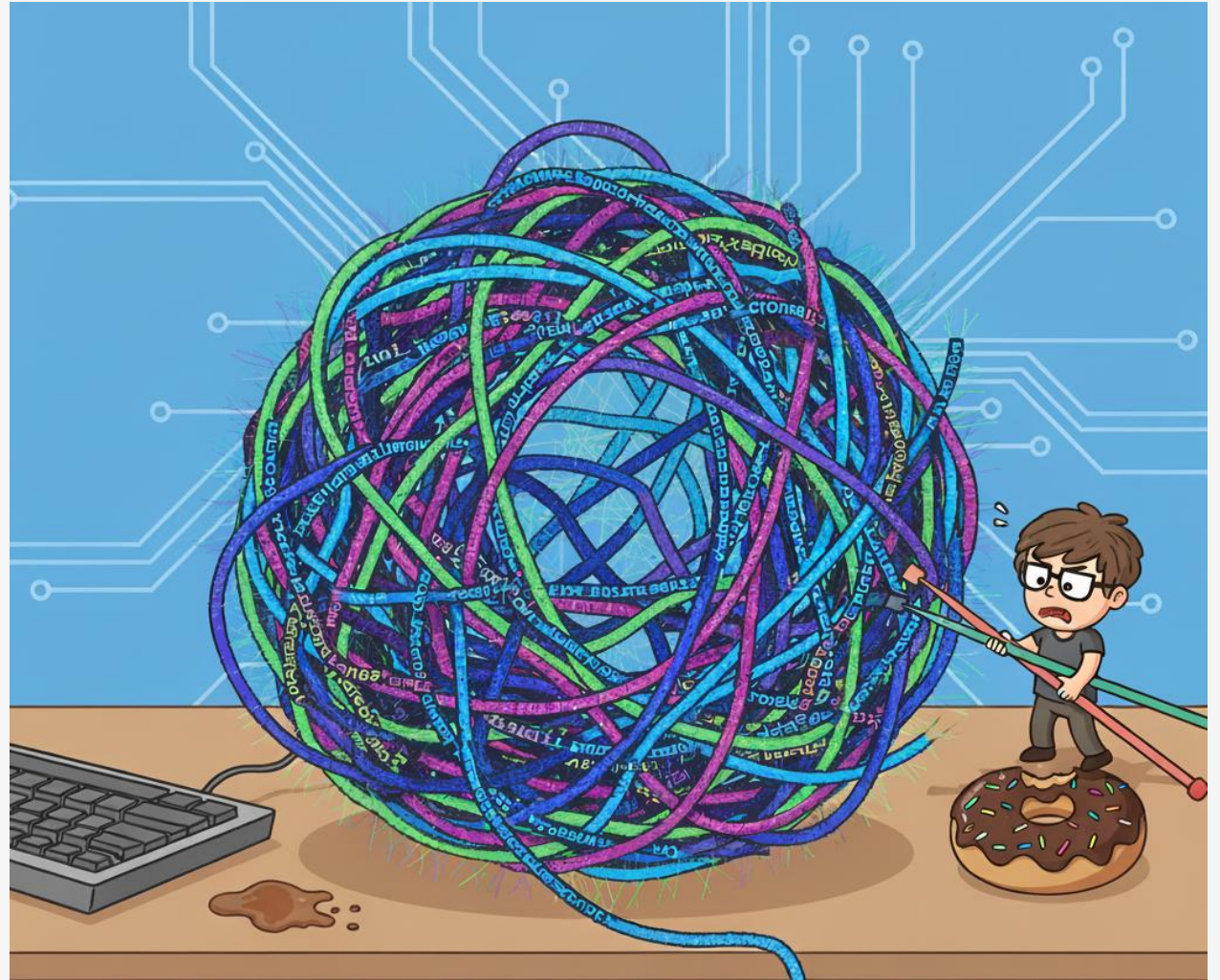
Simple Integration – Legacy Code

```
nanobench::Bench().minEpochIterations(1000000).run("add",  
[&] {  
    result = simple_add(x, y);  
    nanobench::doNotOptimizeAway(result);  
});
```

Simple Integration

– Legacy Code

- Microbenchmarks have pitfalls
- Behaviour changes when not isolated
- Sometimes code can't be isolated
- Be able to measure part of functions



Simple Integration

- Minimal Build System Impact
- Zero or minimal Dependencies
- Broad Compiler Compatibility
- Compatibility with older C++ Standards
- Non-Invasive API:

Dev and Production

- Same Tool
- Be resilient to compiler flags
- Control precisely what is tracked and when

Production

- Export the results (for production)
- Very low overhead
- handles millions of events
- can be easily disabled

Customize

- Easy to add new metrics
- Flexible result storage
- Custom Workflows

Insights

- Find Performance Bottlenecks
- Outlier detection
- Source-level precision
- Call frequency analysis:
- Performance Impact Analysis
- Coverage validation
- Statistical Confidence/Data-Driven Decisions
- Thread Performance Overview

CTRACK

- Single-header library
 - \geq C++17
 - Support all Major Compilers
 - Compiles without Warnings
 - Open source (MIT License)
 - Zero Dependencies
 - CMAKE + Package managers
- github.com/Compaille/ctrack

CTrack

- **Design Philosophy**
- **Minimize Runtime Impact:** Engineered to ensure the application being profiled runs as close to native speed as possible.
- **Prioritize Recording Speed:** The "hot path" of capturing events is extremely lightweight and efficient.
- **Analysis Speed is Secondary:** The "cold path" of calculating statistics can be more computationally intensive, as it occurs after data collection is complete.

CTRACK - Sample

```
void important_function() {  
    auto start = high_resolution_clock::now();  
    // Simulate some work  
    this_thread::sleep_for(milliseconds(100));  
    auto end = high_resolution_clock::now();  
    auto duration = duration_cast<milliseconds>(end - start);  
    cout << "took: " << duration.count() << " ms" << endl;  
}
```

CTRACK - Sample

```
void important_function() {  
    CTRACK; // Just add this one line!  
    // Simulate some work  
    this_thread::sleep_for(milliseconds(100));  
}
```

CTRACK - Sample

```
void important_function() {  
    CTRACK; // Just add this one line!  
    // Simulate some work  
    this_thread::sleep_for(milliseconds(100));  
}
```



```
// Print CTRACK  
ctrack::result_print()
```



Details

filename	function	line	time acc	sd	cv	calls	threads
main.cpp	important_function	6	100.11 ms	0.00 ns	0	1	1
fastest[0-1]%		center[1-99]%					
min	mean	min	mean	med	time a	time ae	
0.00 ns	0.00 ns	100.11 ms	100.11 ms	100.11 ms	100.11 ms	100.11 ms	100.11 ms

Summary

Start		End		time total	time ctracked	time ctracked %
2025-09-13 11:40:51		2025-09-13 11:40:51		100.22 ms	100.11 ms	99.90%
filename	function	line	calls	ae[1-99]%	ae[0-100]%	time ae[0-100]
main.cpp	important_function	6	1	99.90%	99.90%	100.11 ms

CTRACK - Sample

Start	End	Time Total	Time Tracked	Time Tracked %
2025-09-16 00:16:07	2025-09-16 00:16:10	100.16 ms	100.09 ms	99.92%

Filename	Function	Line	Calls	AE[1-99]%	AE[0-100]%	Time AE[0-100]	Time A[0-100]
main.cpp	important_function	8	1	99.92%	99.92%	100.16 ms	100.16 ms

Time Active

```
void slow_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(900));  
}  
void fast_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(100));  
    slow_function();  
}
```

Time Active

```
void slow_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(900));  
}  
void fast_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(100));  
    slow_function();  
}
```

	slow_function	fast_function
Time Active (ms)	900	1000
Time Active (%)	90%	100%

Time Active Exclusive

```
void slow_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(900));  
}  
void fast_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(100));  
    slow_function();  
}
```

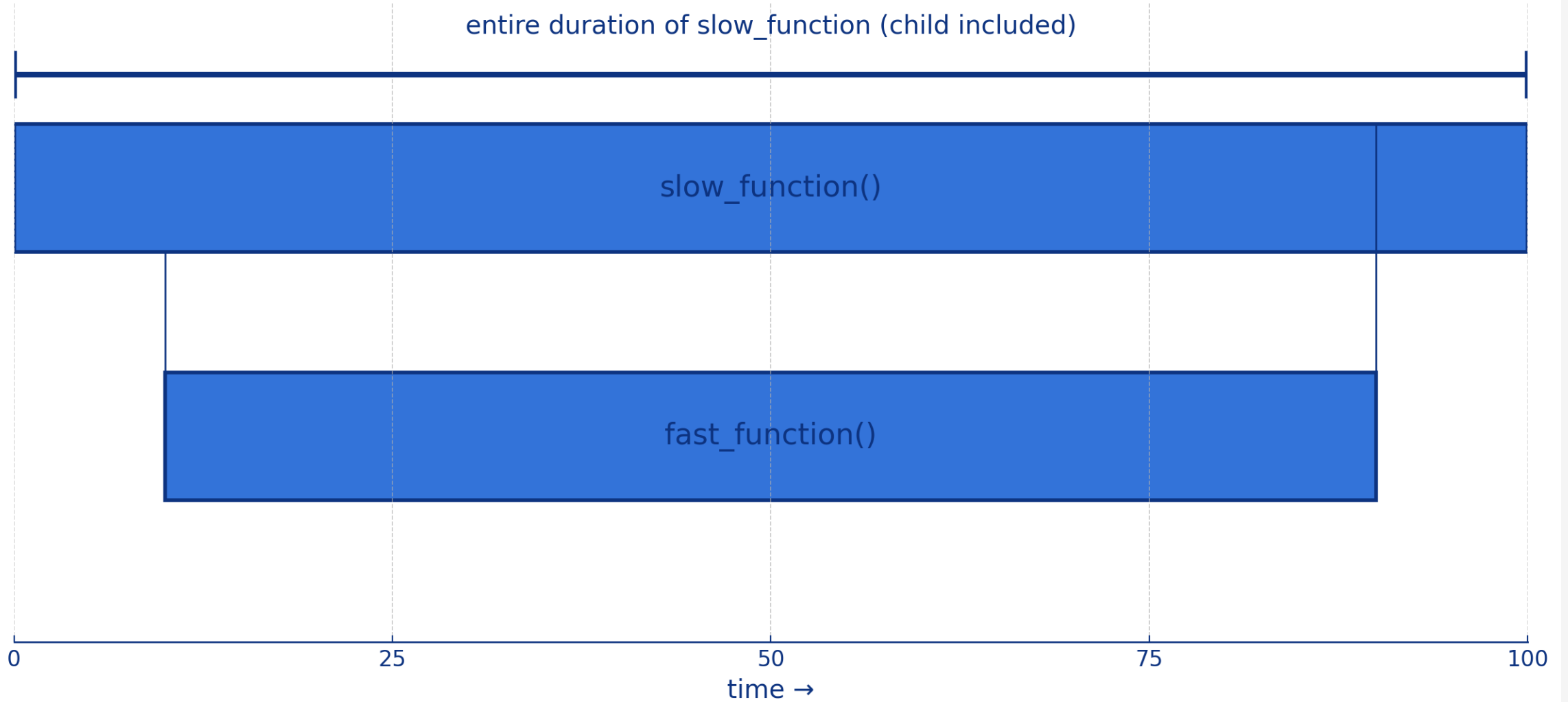
Time Active Exclusive

```
void slow_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(900));  
}  
void fast_function() {  
    CTRACK;  
    this_thread::sleep_for(milliseconds(100));  
    slow_function();  
}
```

	slow_function	fast_function
Time Active-Exclusive (ms)	900	100
Time Active-Exclusive (%)	90%	10%

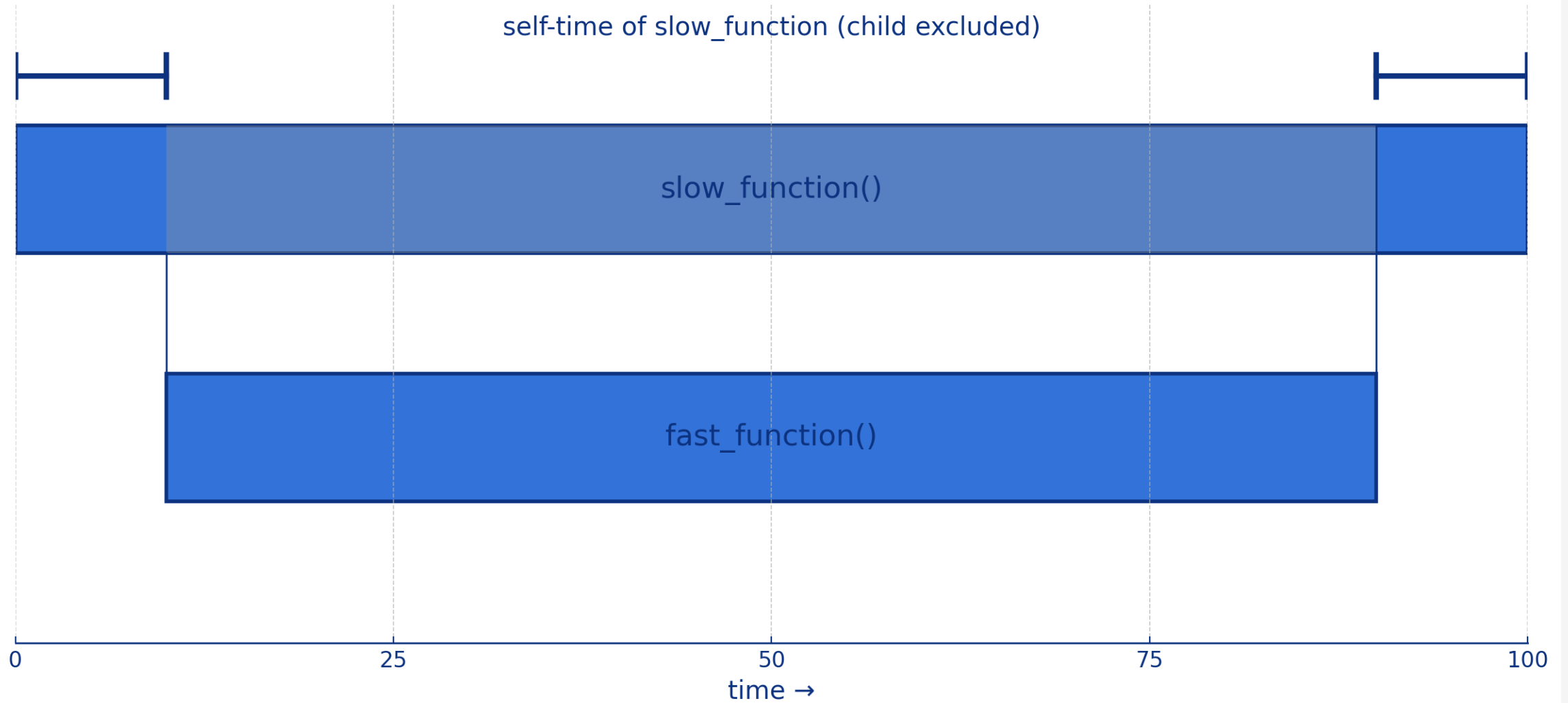
Time Active (inclusive)

entire duration of slow_function (child included)



Time Active-Exclusive

self-time of slow_function (child excluded)



```

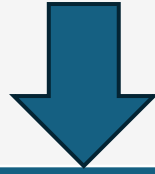
atomic<bool> ready_flag{false};
void child_function() {
    CTRACK;
    this_thread::sleep_for(milliseconds(900));
}
//Thread 2
void worker_thread() {
    CTRACK;
    this_thread::sleep_for(milliseconds(100));
    child_function();
    ready_flag.store(true);
}
//Thread 1
void waiting_thread() {
    CTRACK;
    cout << "[Thread 1] Waiting for ready signal..." << endl;
    while (!ready_flag.load()) {
        this_thread::sleep_for(milliseconds(10));
    }
    cout << "[Thread 1] Got signal! Exiting." << endl;
}

```

Multiple Threads

Filename	Function	Line	Calls	AE[1-99]%	AE[0-100]%	Time AE[0-100]	Time A[0-100]
main.cpp	waiting_thread	34	1	99.94%	99.94%	1.01 s	1.01 s
main.cpp	child_function	12	1	89.30%	89.30%	900.16 ms	900.16 ms
main.cpp	worker_thread	20	1	9.96%	9.96%	100.41 ms	1.00 s

Multiple Threads



Filename	Function	Line	Calls	AE[1-99]%	AE[0-100]%	Time AE[0-100]	Time A[0-100]
main.cpp	waiting_thread	34	1	99.94%	99.94%	1.01 s	1.01 s
main.cpp	child_function	12	1	89.30%	89.30%	900.16 ms	900.16 ms
main.cpp	worker_thread	20	1	9.96%	9.96%	100.41 ms	1.00 s

Why Excluding Outliers Matters: The [1-99]% Range

- Mitigates System Noise
- Isolates Algorithmic Cost
- Establishes a Stable Baseline
- Quantifies Performance Jitter
- Reveals "Hot Path" Performance

Range can be custom

CTRACK - Sample

Filename	Function	Line	Time Acc	SD	CV	Calls	Threads
main.cpp	important_function	8	100.09 ms	0.00 ns	0	1	1

Function	Fastest[0-1]% Min	Fastest[0-1]% Mean	Center[1-99]% Min	Center[1-99]% Mean	Center[1-99]% Med	Center Time A	Center Time AE	Slowest[99-100]% Mean	Slowest[99-100]% Max
variable_timing_function	0.00 ns	0.00 ns	0.00 ns	100.09 ms	100.09 ms	100.09 ms	100.09 ms	0.00 ns	0.00 ns

Detail View

- **Source code location**
- **Calls**
- **Threads**
- **Time Acc**
- **SD (Standard Deviation):**
- **CV (Coefficient of Variation):** Normalized variability (SD/mean)

- **Timing Distribution Table**
 - **Fastest [0-1]%**: The best-case performance
 - **Center [1-99]%**: Typical performance
 - **Slowest [99-100]%**: Worst-case performance
- (Min Mean Max)

Accuracy

Example & Comparison with sample based profiler

```
// Fast function called many times - often missed by sampling
void micro_operation()
{
    CTRACK;
    this_thread::sleep_for(milliseconds(1));
}
```

```
// Function with high variance - sampling misses the pattern
void variable_timing_function(bool slow_path)
{
    CTRACK;
    if (slow_path)
    {
        this_thread::sleep_for(milliseconds(100));
    }
    else
    {
        this_thread::sleep_for(milliseconds(1));
    }
}
```

```
int main()
{
    const int micro_op_count = 1000;

    for (int i = 0; i < micro_op_count; ++i)
    {
        micro_operation();
    }

    for (int i = 0; i < 100; ++i)
    {
        bool slow = i % 10 == 0; // Every 10th call is slow
        variable_timing_function(slow);
    }
    return 0;
}
```

```
int main()
{
    const int micro_op_count = 1000;

    for (int i = 0; i < micro_op_count; ++i)
    {
        micro_operation();
    }

    for (int i = 0; i < 100; ++i)
    {
        bool slow = i % 10 == 0; // Every 10th call is slow
        variable_timing_function(slow);
    }
    return 0;
}
```

~ 1 second expected

~ 1 second expected

Sample-Based Profiling

	40	<code>// Pattern 1: Many fast calls (total ~1 second)</code>
	41	<code>cout << "Running " << micro_op_count << " micro-operations..." << endl;</code>
	42	<code>for (int i = 0; i < micro_op_count; ++i)</code>
	43	<code>{</code>
16 (53,33 %)	44	<code>micro_operation();</code>
	45	<code>}</code>
	46	
	47	<code>// Pattern 3: Variable timing (90% fast, 10% slow)</code>
	48	<code>cout << "Running 100 variable timing calls (10% slow)..." << endl;</code>
	49	
	50	<code>for (int i = 0; i < 100; ++i)</code>
	51	<code>{</code>
	52	<code>bool slow = i % 10 == 0; // Every 10th call is slow</code>
3 (10,00 %)	53	<code>variable_timing_function(slow);</code>
	54	<code>}</code>
	55	

Sample-Based Profiling

```
19 // Function with high variance - sampling misses the pattern
20 void variable_timing_function(bool slow_path)
21 {
22     CTRACK;
23     if (slow_path)
24     {
25         // Slow path: 100ms
26         this_thread::sleep_for(milliseconds(100));
27     }
28     else
29     {
30         // Fast path: 1ms
31         this_thread::sleep_for(milliseconds(1));
32     }
33 }
```

1 (4,35 %)

sampling limitations

CTRACK

Start	End	Time Total	Time Tracked	Time Tracked %
2025-09-16 00:16:07	2025-09-16 00:16:10	2.17 s	2.17 s	99.97%

Filename	Function	Line	Calls	AE[1-99]%	AE[0-100]%	Time AE[0-100]	Time A[0-100]
main.cpp	variable_timing_function	22	100	46.39%	51.07%	1.11 s	1.11 s
main.cpp	micro_operation	15	1000	47.48%	48.90%	1.06 s	1.06 s

CTRACK

Filename	Function	Line	Time Acc	SD	CV	Calls	Threads
main.cpp	variable_timing_function	22	1.11 s	29.71 ms	2.68174	100	1
main.cpp	micro_operation	15	1.06 s	153.01 µs	0.144228	1000	1

Function	Fastest[0-1]% Min	Fastest[0-1]% Mean	Center[1-99]% Min	Center[1-99]% Mean	Center[1-99]% Med	Center Time A	Center Time AE	Slowest[99-100]% Mean	Slowest[99-100]% Max
variable_timing_function	1.03 ms	1.03 ms	1.03 ms	10.27 ms	1.11 ms	1.01 s	1.01 s	100.42 ms	100.42 ms
micro_operation	1.02 ms	1.02 ms	1.02 ms	1.05 ms	1.04 ms	1.03 s	1.03 s	2.05 ms	4.31 ms

Customization

Customization

- Custom naming

```
void example_function() {  
    CTRACK_NAME("my custom name");  
}
```

Customization

- DEV and PROD

```
void example_function() {  
    CTRACK_DEV;  
}  
void important_function() {  
    CTRACK_PROD;  
}
```

Customization

- „Sub“ Functions via Scopes

```
void complex_algorithm() {  
    CTRACK; // Track the whole function  
    {  
        CTRACK_NAME("algorithm_init");  
        this_thread::sleep_for(milliseconds(10));  
    }  
    for (int i = 0; i < 3; ++i) {  
        CTRACK_NAME("algorithm_iteration");  
        this_thread::sleep_for(milliseconds(15));  
    }  
}
```

Customization

```
ctrack::ctrack_result_settings custom_settings;  
custom_settings.non_center_percent = 2; // Use [2-98]  
custom_settings.min_percent_active_exclusive = 1.0;  
custom_settings.percent_exclude_fastest_active_exclusive = 0.0;
```

Customization

- Different outputs
- Direct Data Access (ctrack_result_tables)

Real World Example

filename	function	line	calls	ae[1-99]%
src/camera_capture.cpp	capture_manager::grab_frames	118	12	42.80%
src/preprocess.cpp	resize_bgrx_to_rgb	63	12	11.30%
src/fusion.cpp	depth_temporal_sync_and_fuse	142	12	7.70%
src/segmentation.cpp	segment_net::infer	332	12	6.40%
src/tracking.cpp	kcf_tracker::update	175	12	4.70%
src/camera_capture.cpp	dma_copy_to_userspace	247	12	9.00%
src/io_writer.cpp	mjpeg_writer::write_frame	89	12	3.20%

filename	function	line	calls	ae[1-99]%
src/segmentation.cpp	segment_net::infer	332	12	38.00%
src/preprocess.cpp	resize_bgrx_to_rgb	63	12	12.70%
src/camera_capture.cpp	capture_manager::grab_frames	118	12	9.30%
src/fusion.cpp	depth_temporal_sync_and_fuse	142	12	6.90%
src/postprocess.cpp	mask_morphological_ops	77	12	2.70%
src/tracking.cpp	deepsort_tracker::update	264	12	1.80%
src/camera_capture.cpp	dma_copy_to_userspace	247	12	1.10%

Implementation Details

```
struct Event
{
    time_point start_time;
    time_point end_time;
    std::string_view filename;
    std::string_view function;
    int line;
    int thread_id;
    unsigned int event_id;
}
```

How does CTRACK work

- **RAII-Based Timing** Uses C++ object lifetime for automatic measurement
- **Constructor = Start:** Timer begins when CTRACK object is created
- **Destructor = Stop** Timer ends when object goes out of scope
- **Zero Manual Intervention**:** No need to manually stop timers
- **Exception Safe** Destructor always called, even with exceptions
- **Minimal Overhead**

```
void my_function() {  
    CTRACK;  
    // Expands to:  
    EventHandler _ctrack_instance{__LINE__,  
    __FILE__, __FUNCTION__};  
  
    // Your code here  
    if (error) return;  
    // Destructor called here  
  
    // More code  
} // Destructor called here
```

Key Advantages Over Manual Timing

1. **Automatic:** No need to manually stop timers
2. **Complete:** Captures all exit paths
3. **Clean:** One line of code vs multiple timing statements
4. **Safe:** Exception-safe by design
5. **Fast:** Optimized for production use

```
void important_function() {  
    auto start = high_resolution_clock::now();  
    // Simulate some work  
    this_thread::sleep_for(milliseconds(100));  
    auto end = high_resolution_clock::now();  
    auto duration =  
        duration_cast<milliseconds>(end - start);  
    cout << "took: " << duration.count() << "  
    ms" << endl;  
}
```

Per-Thread Data Buffers: Zero Contention

```

// Global store with per-thread vectors
struct store {
    inline static std::deque<t_events> a_events{};    // One vector per thread
    inline static std::mutex event_mutex;           // Used ONLY during
setup
};

// Thread-local pointers to this thread's buffer
inline thread_local t_events* event_ptr = nullptr;

// First-time thread setup (happens once on thread spawn)
int fetch_event_t_id() {
    if (thread_id == nullptr) {
        std::scoped_lock lock(store::event_mutex);    // ONLY lock
        store::a_events.emplace_back(t_events{});    // Create buffer
        event_ptr = &store::a_events[*thread_id];    // Point to it
        event_ptr->reserve(100);                      // Pre-allocate
    }
    return *thread_id;
}

// Recording (no locks!)
event_ptr->emplace_back(Event{...});    // Just append to vector

```

Per-Thread Data Buffers: Zero Contention

- Thread-Local Pointers: Each thread “owns” its data structures
- One-Time Setup: Single mutex lock per thread lifetime
- Lock-Free Hot Path: Recording happens without synchronization
- Perfect Scalability: N threads = $N \times$ throughput

Implicit Call Graph Construction

- Stack Mimicry via RAll: Each scope maintains parent-child relationship
- Zero-Lock Recording: No mutexes during event capture
- Thread-Local Current Event: Each thread tracks its own call stack
- Automatic Hierarchy: Destructor order guarantees correct nesting

```

// Constructor - Entering a scope
EventHandler(...) {
    previous_event_id = *current_event_id; // Save parent
    event_id = ++(*current_event_cnt); // Get new ID
    *current_event_id = event_id; // Become current
}

// Destructor - Leaving a scope
~EventHandler() {
    event_ptr->emplace_back(Event{...}); // Record self
    *current_event_id = previous_event_id; // Restore parent

    if (previous_event_id > 0) {
        // Register as child of parent
        (*sub_events_ptr)[previous_event_id].push_back(event_id);
    }
}

```

Implicit Call Graph Construction

- Root → Branch → Leaf hierarchy automatically captured
- time ae (active exclusive) shows time minus children
- Perfect parent-child relationships with zero explicit tracking

How does CTRACK work

- C++17 Parallel Algorithms (par_unseq) for fast statistics calculation
- Interval Merging: Solves multithread time overlap problem
- Versioned Namespaces: ABI safety without sacrificing ergonomics

Parallel Analysis

```
constexpr auto execution_policy =  
std::execution::par_unseq;  
  
return std::transform_reduce(  
    execution_policy, // Parallel execution  
    vec.begin(), vec.end(),  
    FieldType{}, std::plus<>(),  
    [field](const auto &item) { return item.*field;  
});
```

Interval Merging

```
// Problem: 4 threads × 10ms ≠ 40ms wall time!
sorted_create_grouped_simple_events(events) {

    for (size_t i = 1; i < events.size(); i++) {

        if (result[current].end_time >= events[i].start_time) {

            // Merge overlapping intervals
            result[current].end_time =
                max(result[current].end_time, events[i].end_time);
        } else {
            result.push_back(events[i]); // New interval
        }
    }
}
```

Versioned Namespaces

```
#define CTRACK_VERSION_NAMESPACE v1_1_0
namespace ctrack {
    inline namespace CTRACK_VERSION_NAMESPACE {
        // All API here
    }
}
// Users write: ctrack::EventHandler
// Actually get: ctrack::v1_1_0::EventHandler
```

Impact

Adoption and Use Cases

Open Source

- Commitment to Maintenance
- Responsibility
- Performance
- Accuracy

CTRACK Benchmark Suite

- Accuracy Measurement: Validates timing precision against known values
- Overhead Analysis: Measures performance impact
- Memory Efficiency: Tracks bytes per event
- Scalability Testing
- Baseline Comparison Regression detection across releases

CTRACK Test Suite

- 6 Test Categories: Basic, Multithreaded, Nested, Statistics, Results, Edge Cases
- Doctest Framework: Modern C++ testing with subcases
- 100+ Test Cases: Every feature validated
- Thread Safety: Validates concurrent access patterns

Roadmap

- Flamegraph
- JSON
- CI/CD Runners
- RDTSC
- Light Version
- .. Your ideas!

Contribution

- **Submit Pull Requests:** For bug fixes, new features, or performance improvements.
- **Report Issues:** Help us find and fix bugs by reporting them on GitHub.
- **Request Features:** Share your ideas for how to make CTRACK even better.
- **Tackle the Roadmap:** Contribute to planned features like JSON and SQL support.



The Ultimate Bottleneck

- → It's us
- We can always replace code.
- We can never replace people.

Q & A



Any Questions?

Grischa Hauser
CEO of COMPAILE Solutions
GmbH

www.compaille.com

Grischa.hauser@compaille.com