EUROPEAN
SPALLATION
SOURCE

Code Performance Analysis and Visualisation

A blaze through performance tools on macOS

PRESENTED BY MORTEN HILKER-SKAANING

2020-09-13

Agenda

1. macOS Instruments: CPU time sampler
2. macOS Instruments: Viewing transient allocations
3. Mention Callgrind + QCachegrind
4. Understanding machine code using Ghidra
5. Fast code with ISPC language + compiler
6. Viewing large C++ projects with SourceTrail
7. Viewing C++ compilation time profiles

1

macOS Instruments: CPU time sampler

macOS Instruments: CPU time sampler

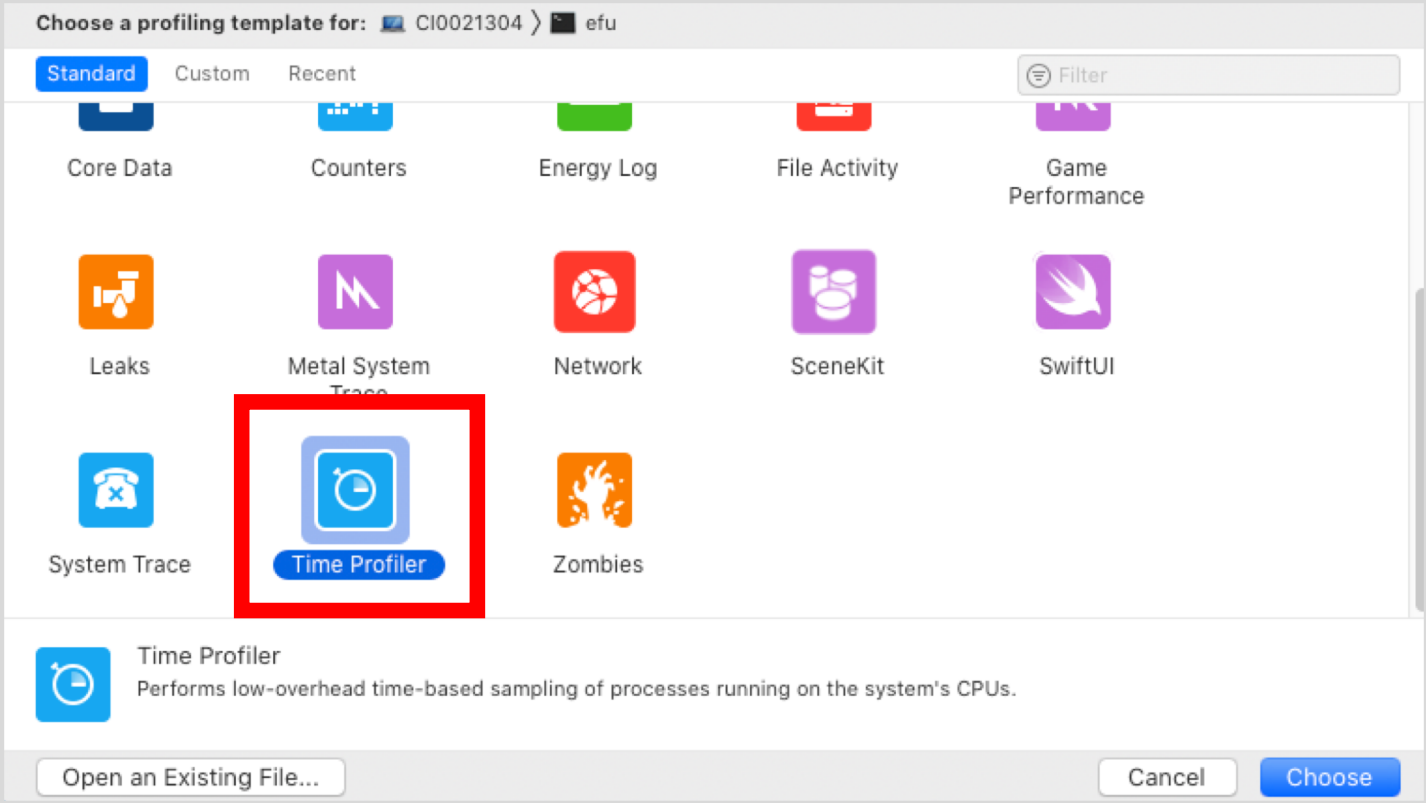
Install

Easier than Cachegrind+QCachegrind

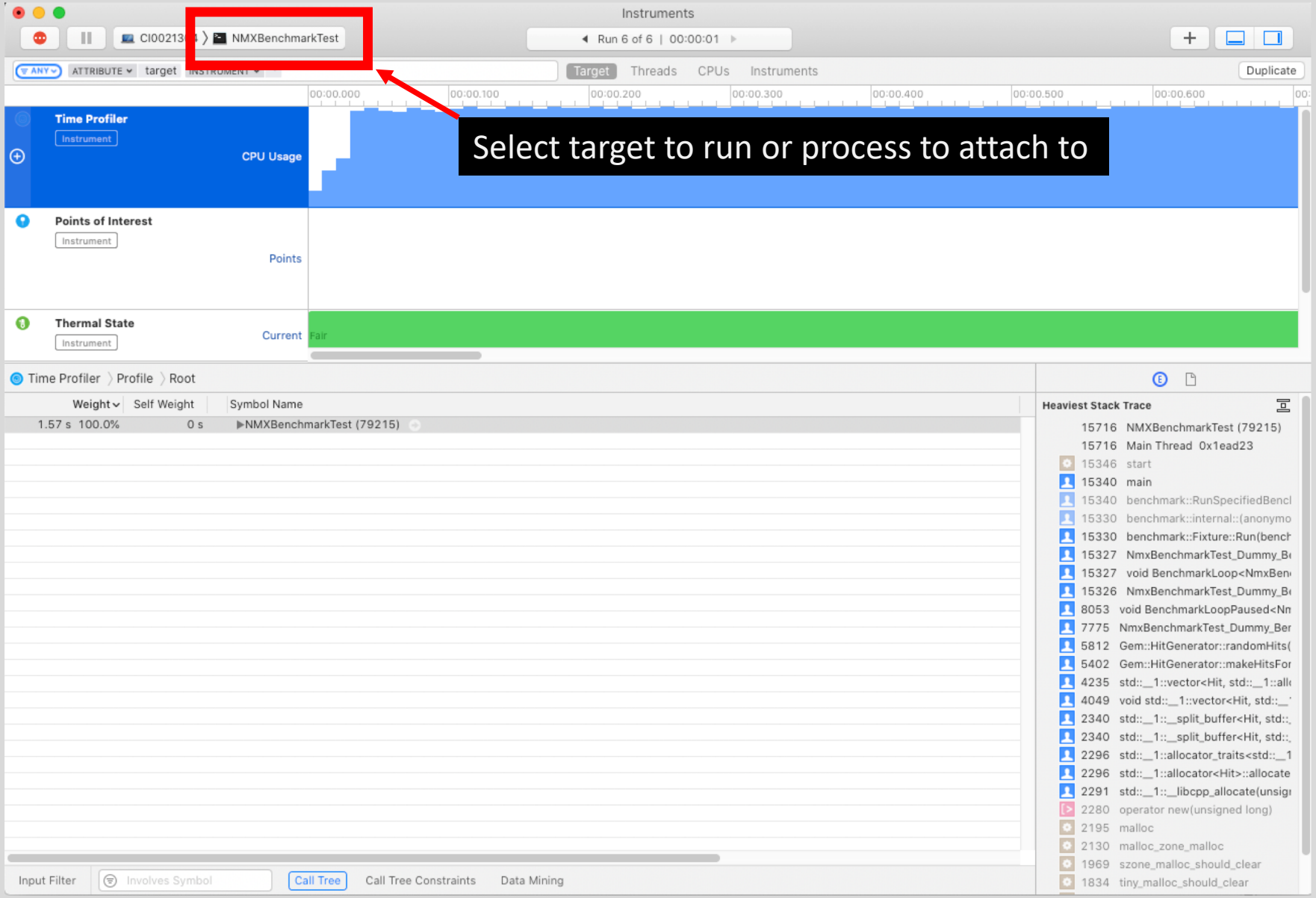
Bundled with Xcode, worth the download

Case: Allocations in NMX benchmark

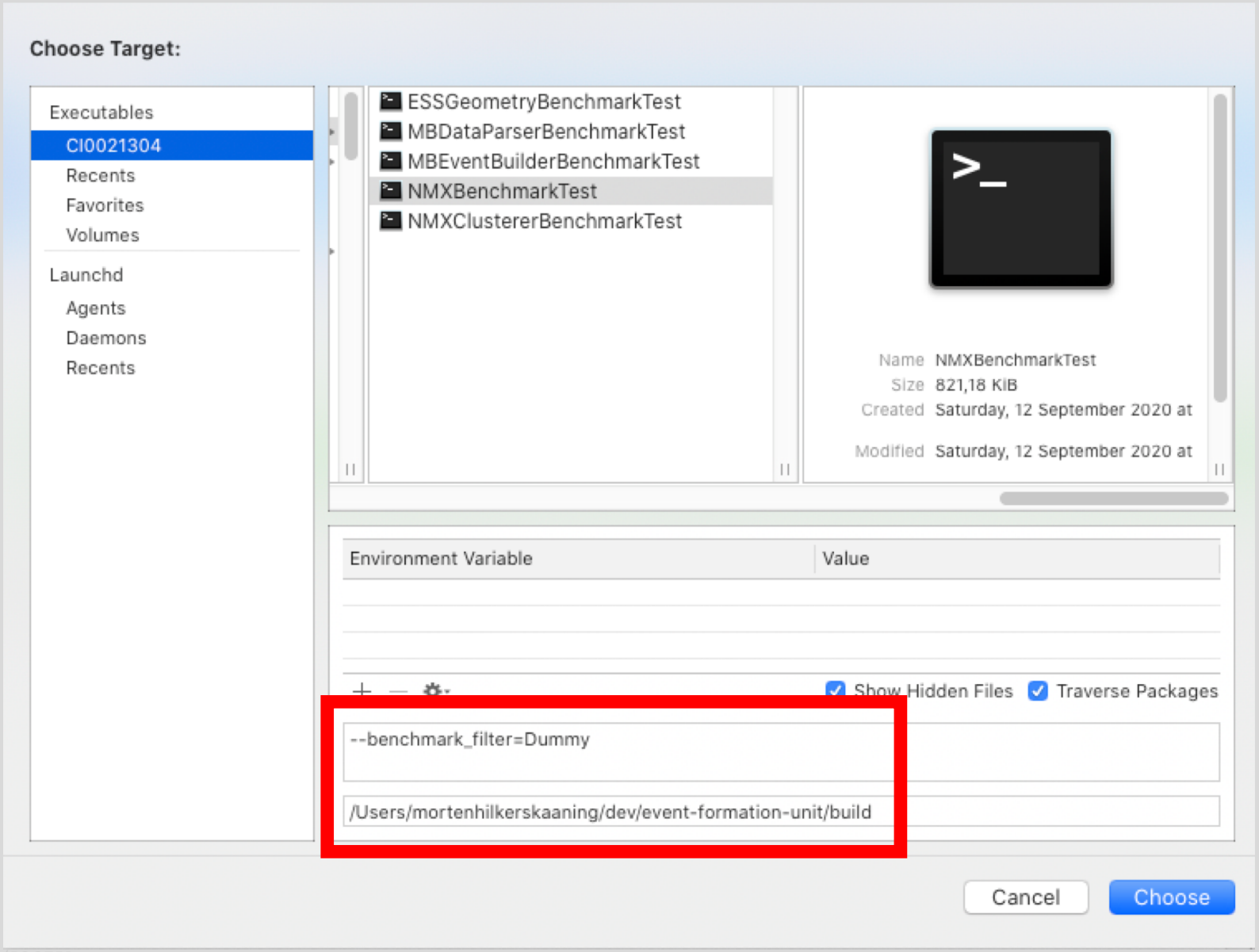
macOS Instruments: CPU time sampler



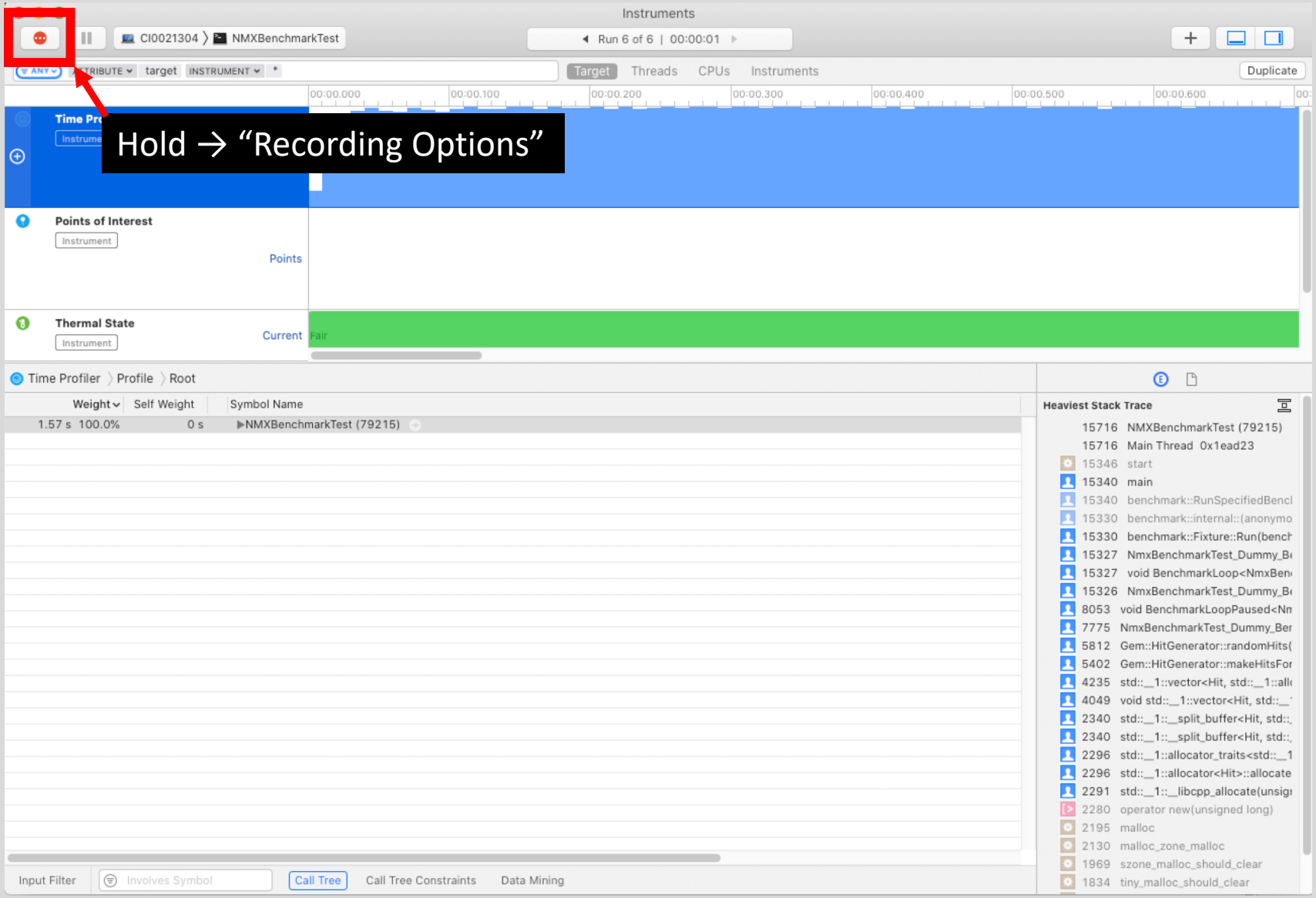
macOS Instruments: CPU time sampler



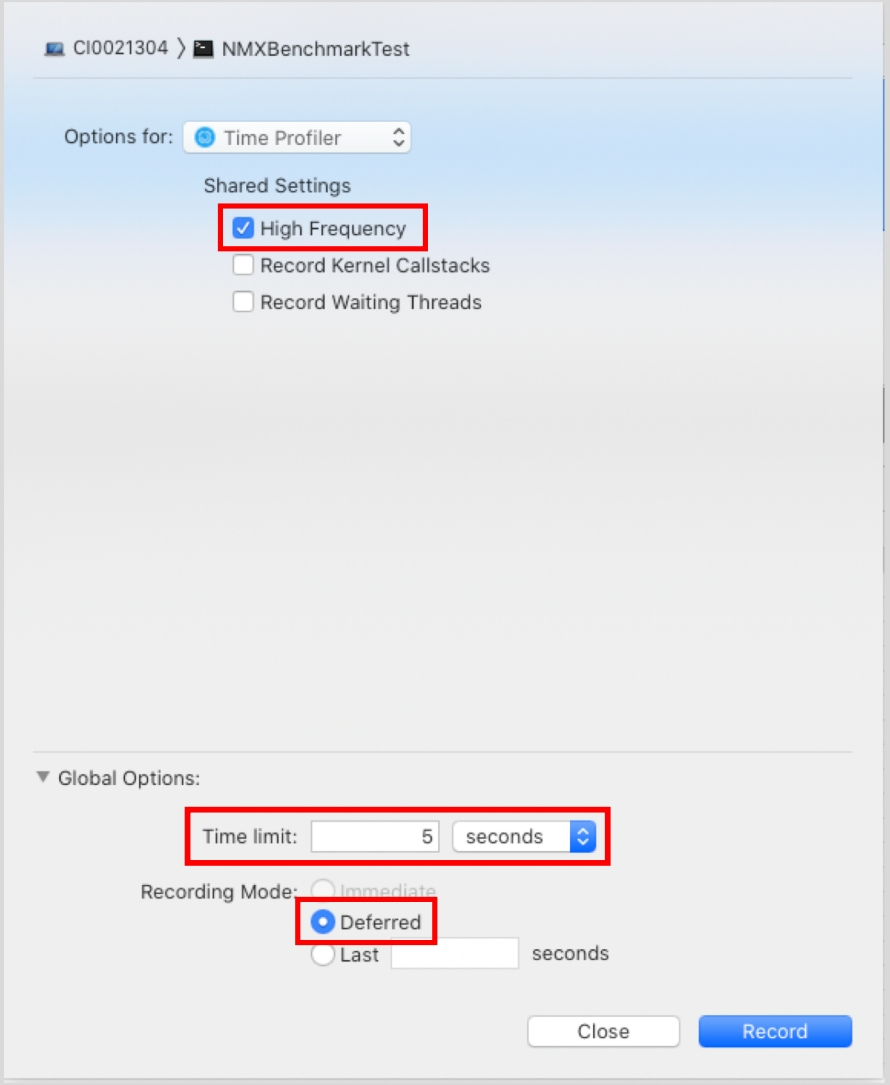
macOS Instruments: CPU time sampler



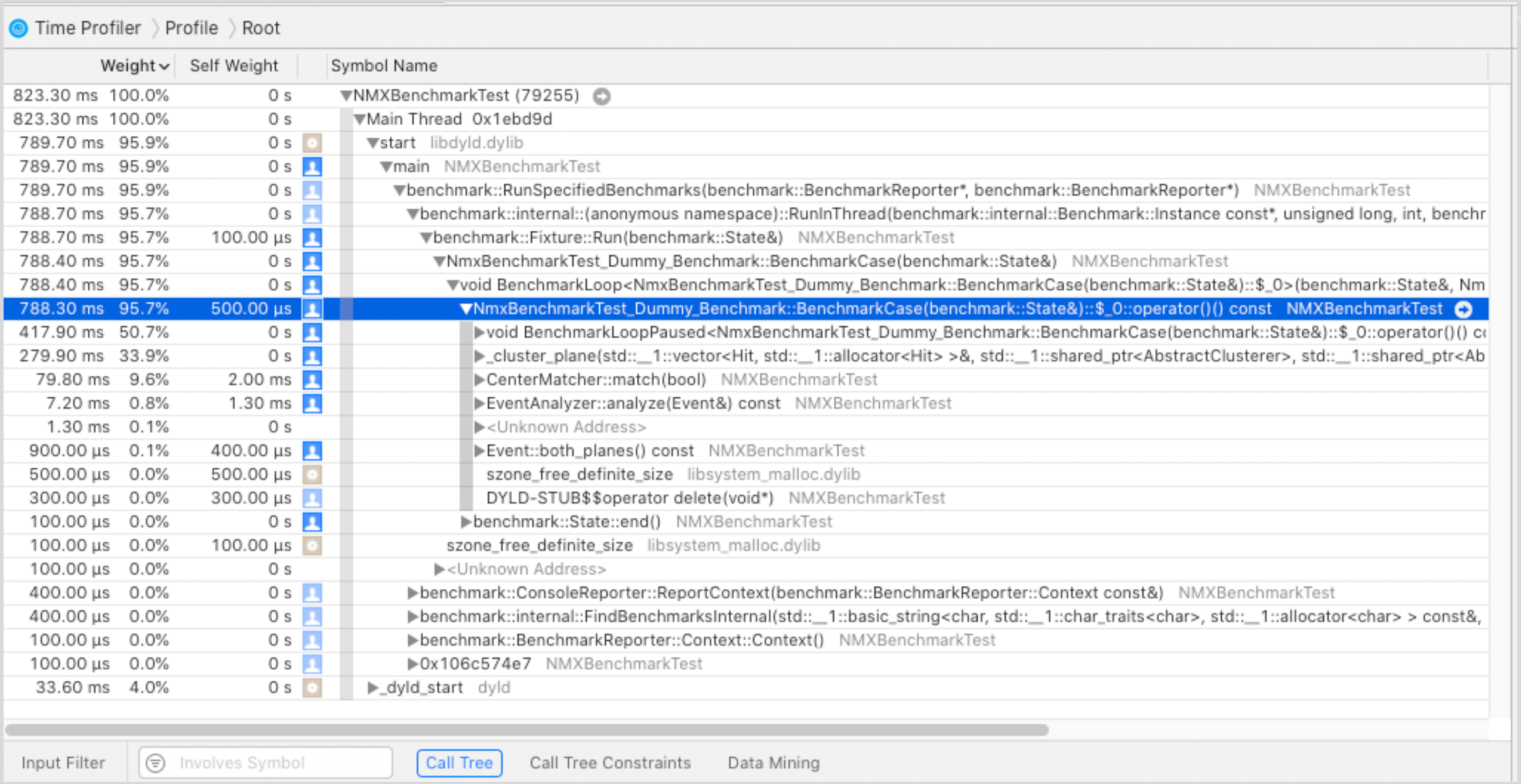
macOS Instruments: CPU time sampler



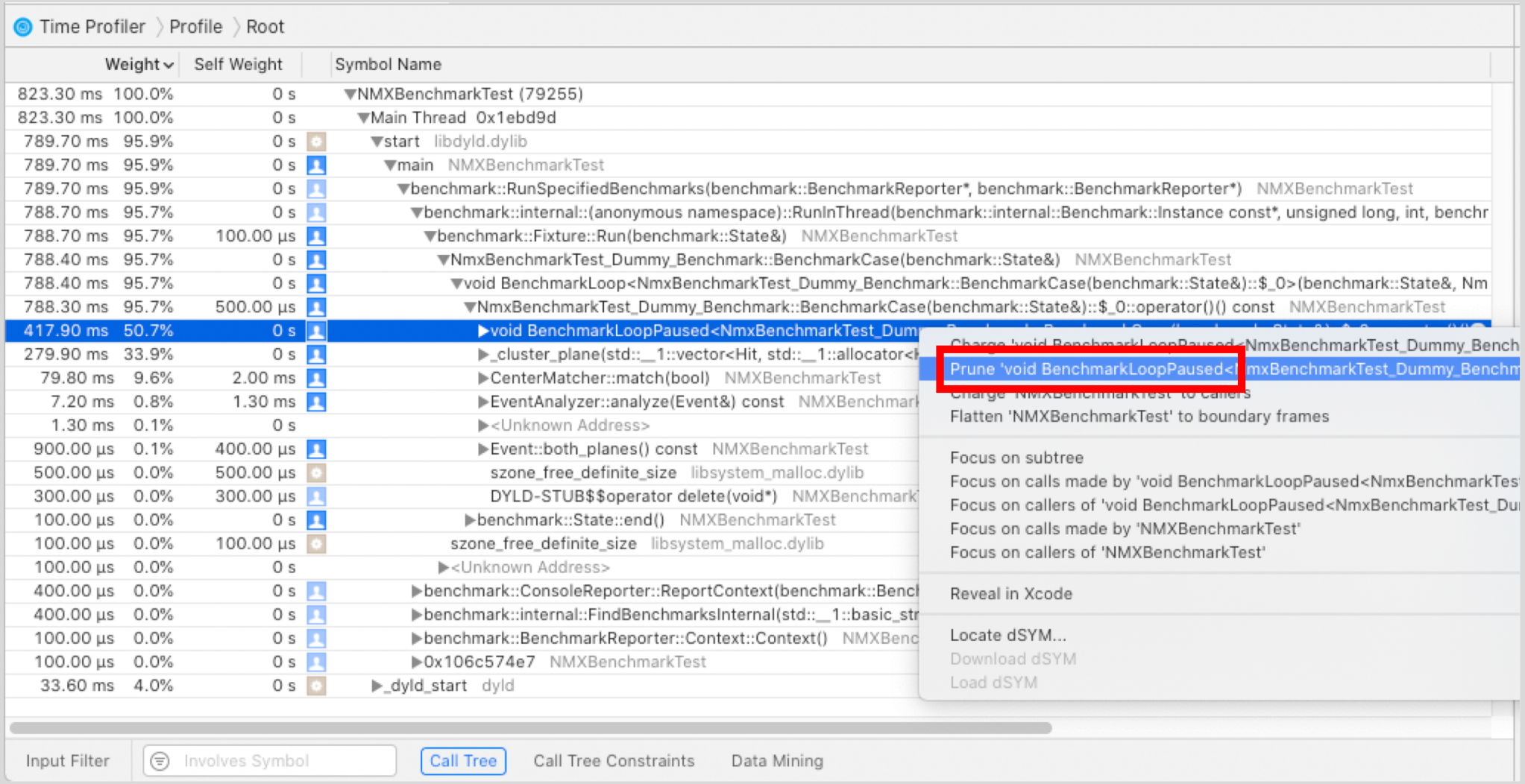
macOS Instruments: CPU time sampler



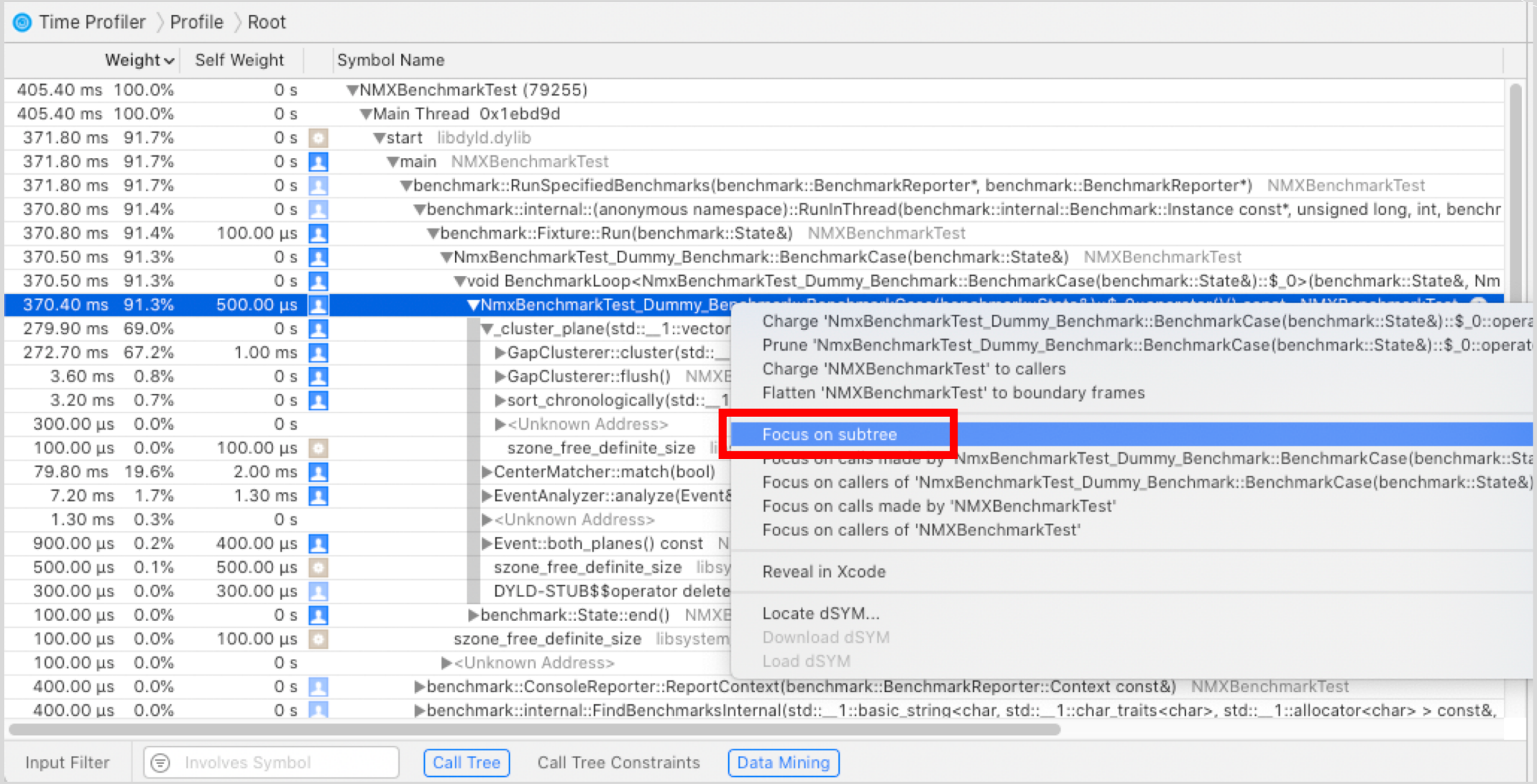
macOS Instruments: CPU time sampler



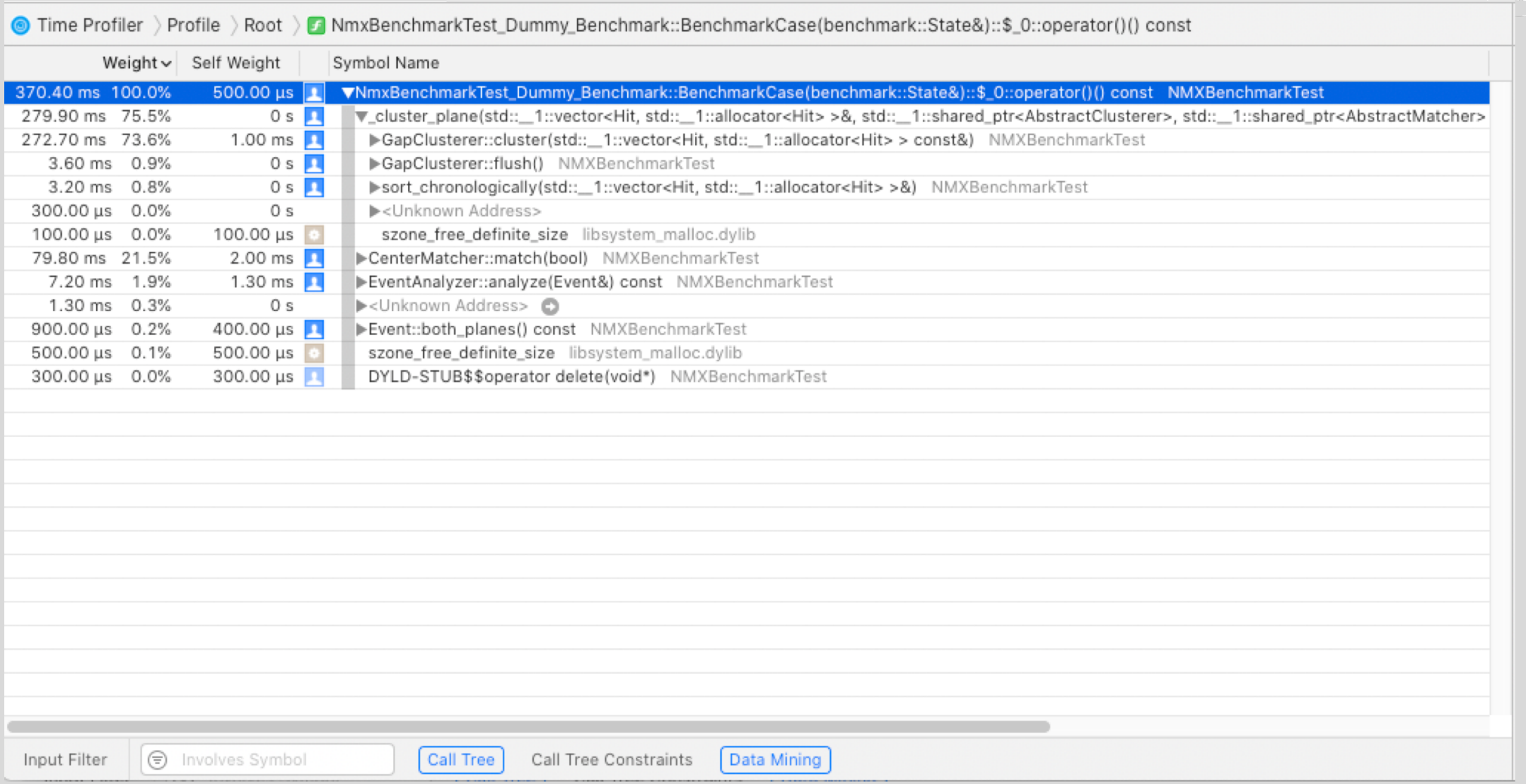
macOS Instruments: CPU time sampler



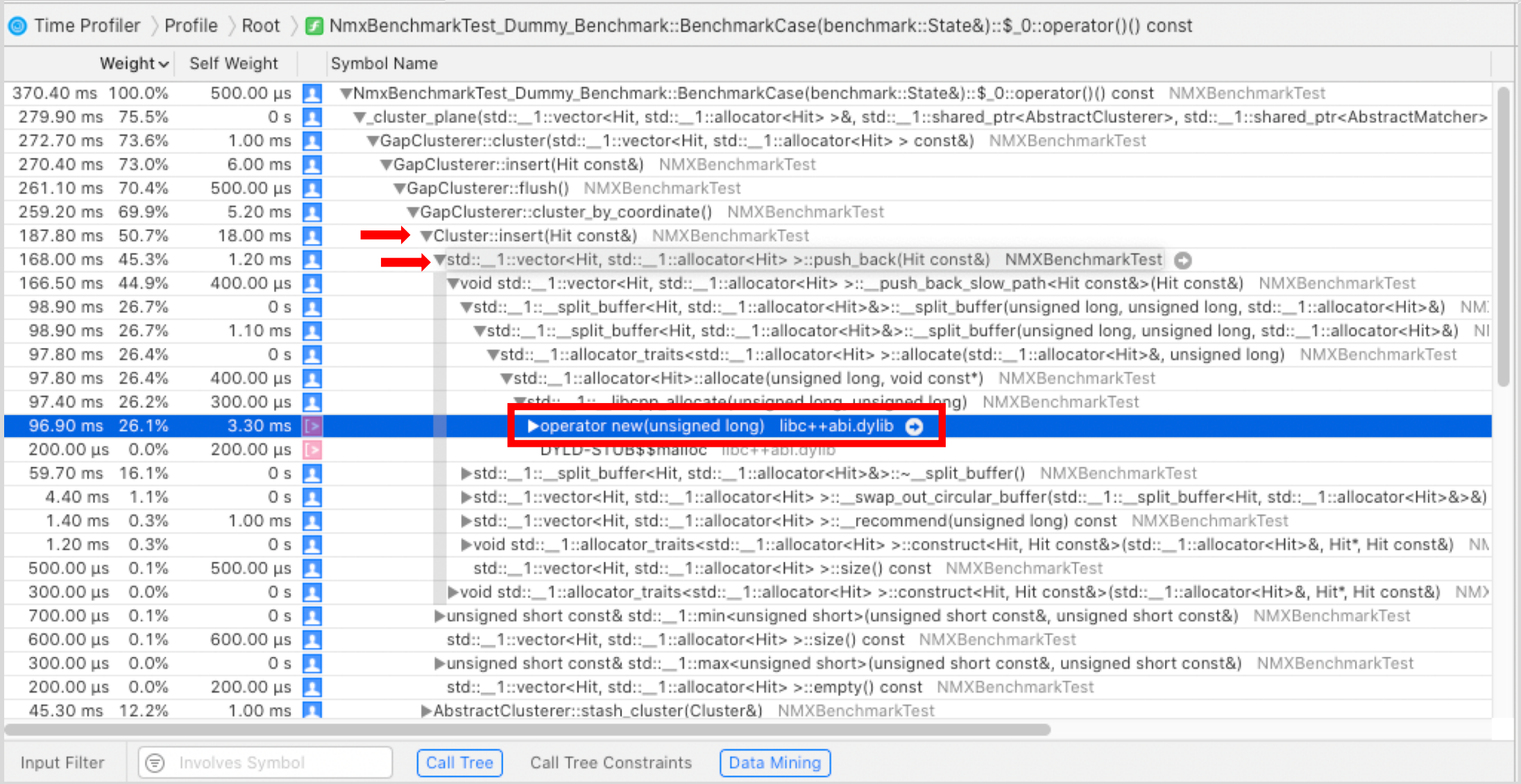
macOS Instruments: CPU time sampler



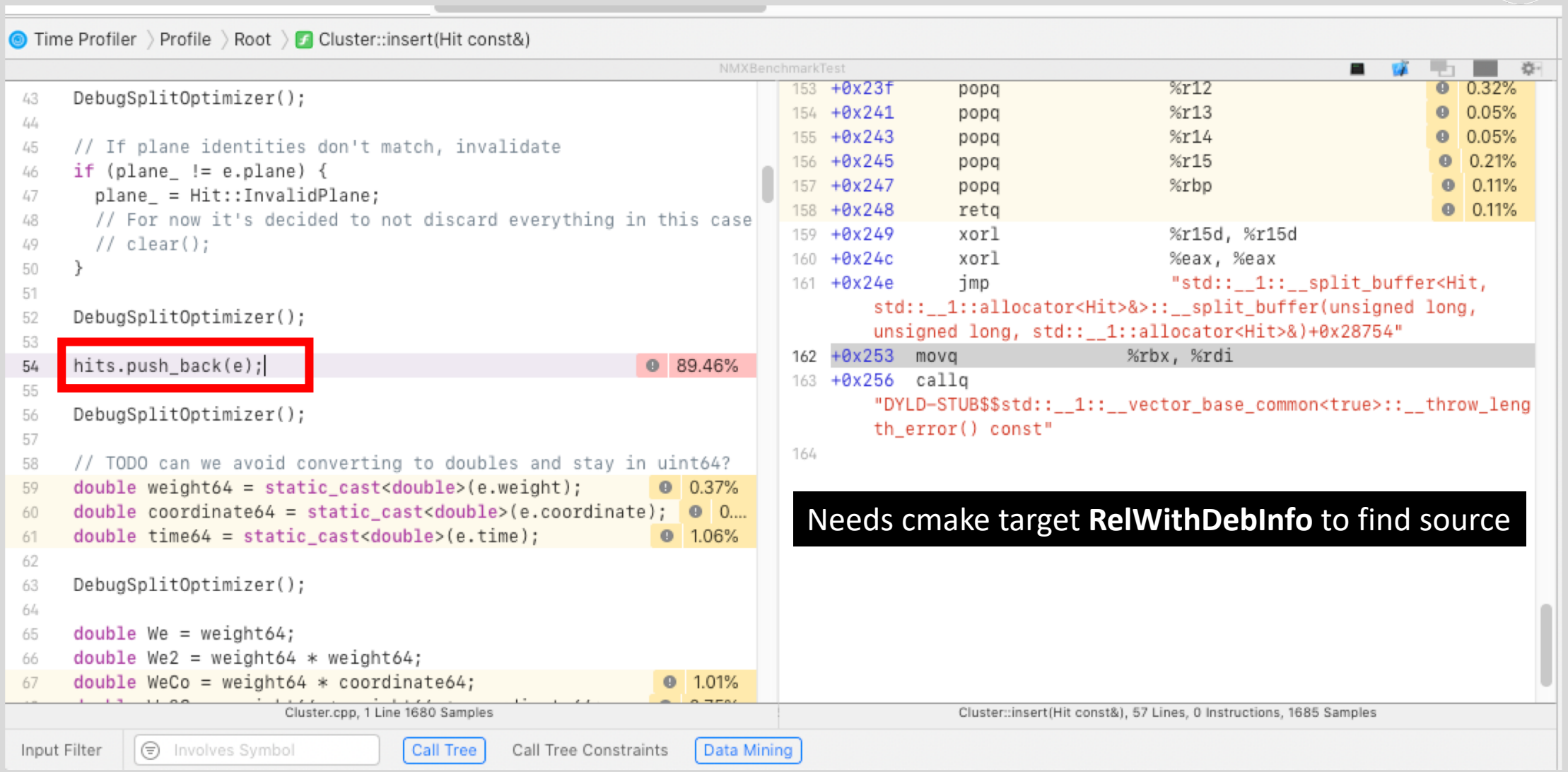
macOS Instruments: CPU time sampler



macOS Instruments: CPU time sampler



macOS Instruments: CPU time sampler



Needs cmake target **RelWithDebInfo** to find source

macOS Instruments: CPU time sampler

Debugging optimizations

Try prevent optimizer from reordering sections of code

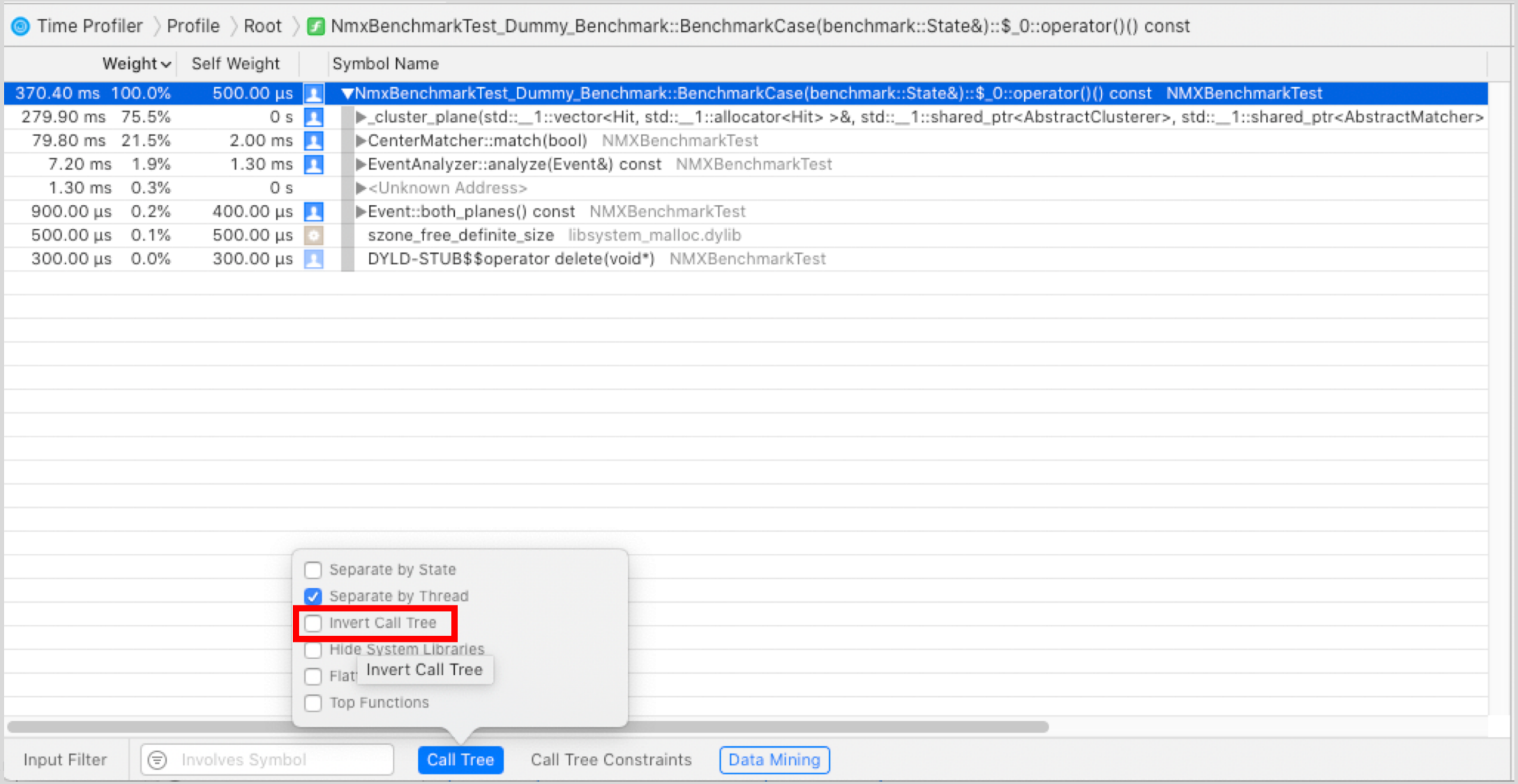
Need to understand generated code from individual sections

```
__attribute__((noinline)) void DebugSplitOptimizer() {
  asm volatile("" : : : "memory");
}
```

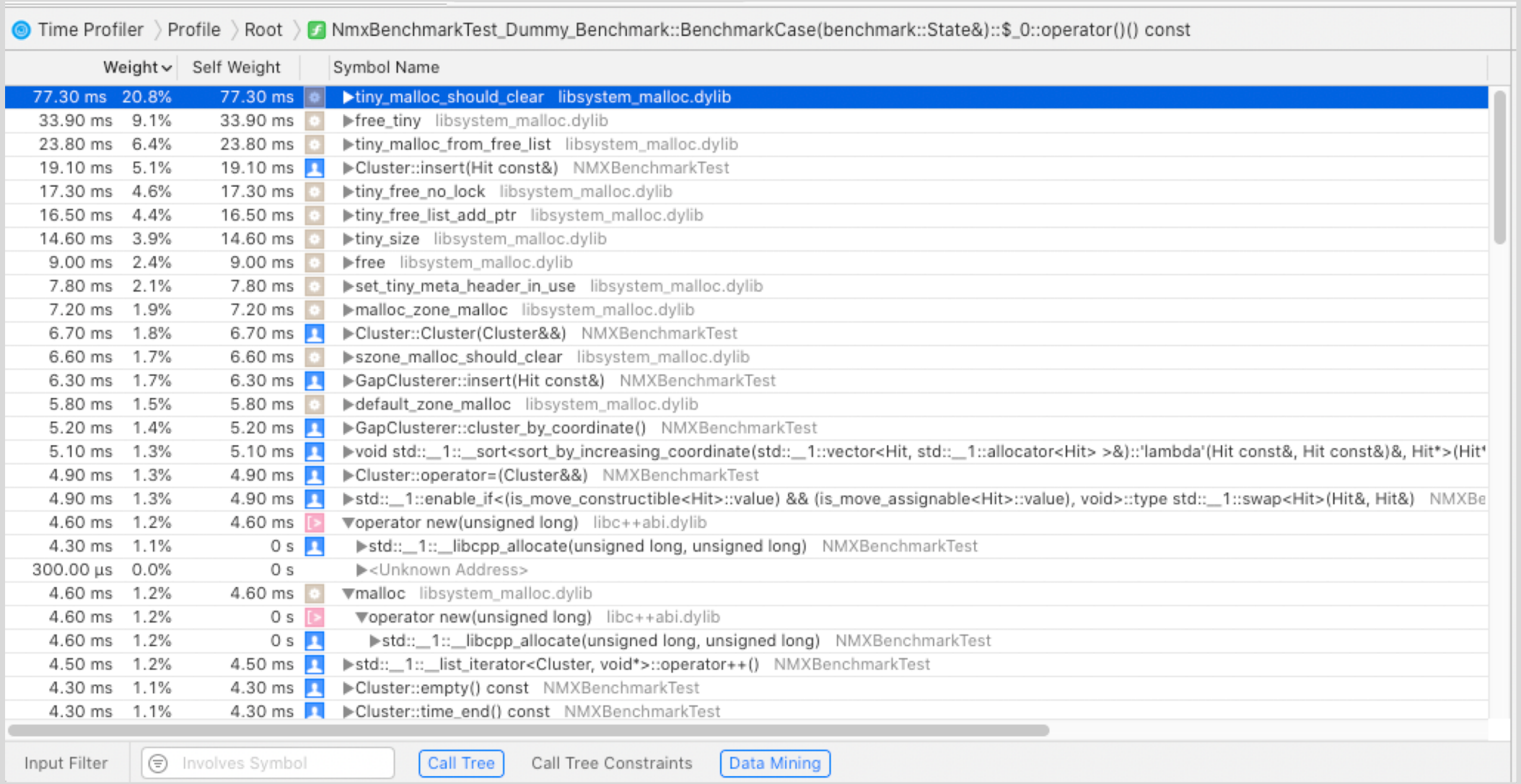
Code taken from Google Benchmark: `::benchmark::ClobberMemory()`

Article by Mike Action

macOS Instruments: CPU time sampler



macOS Instruments: CPU time sampler



macOS Instruments: CPU time sampler

Conclusion

Simple tool to setup and use for iterations

Wrote custom allocator for std::vector and std::list → 5x perf

2

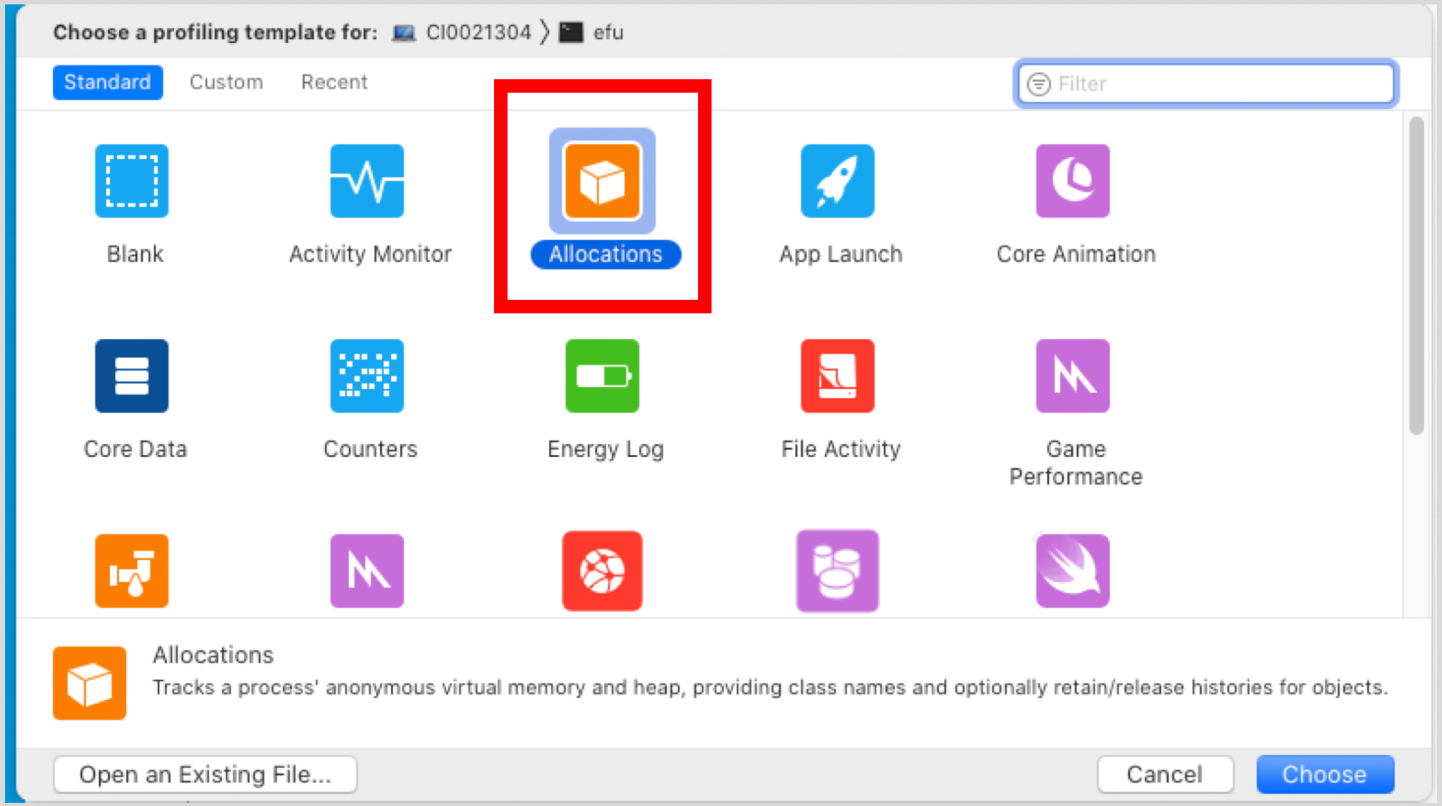
macOS Instruments: Viewing transient allocations

macOS Instruments: Transient allocations

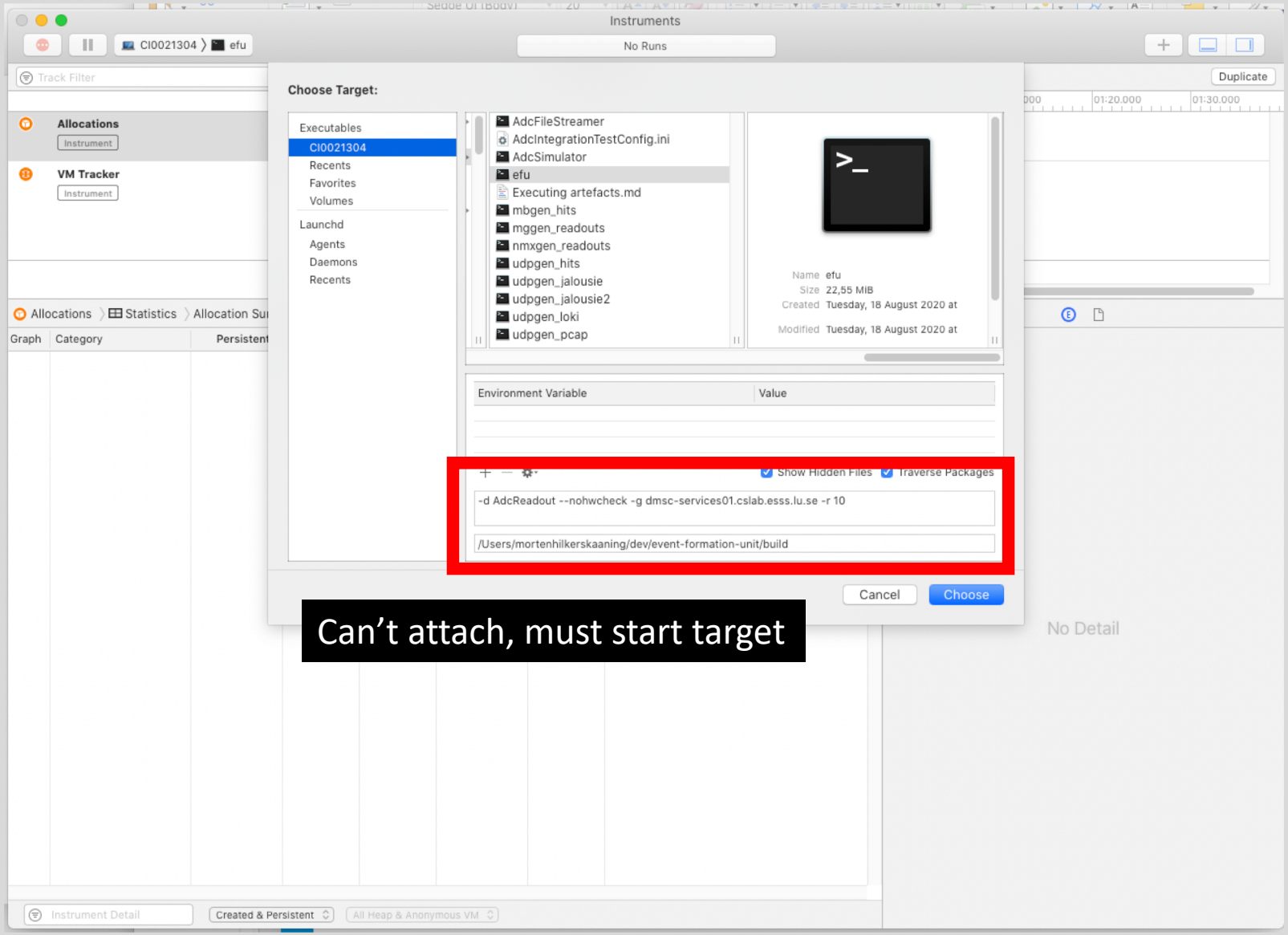
A simple way to view short-lived allocations

Useful when knowing you spend time on allocations

macOS Instruments: Transient allocations

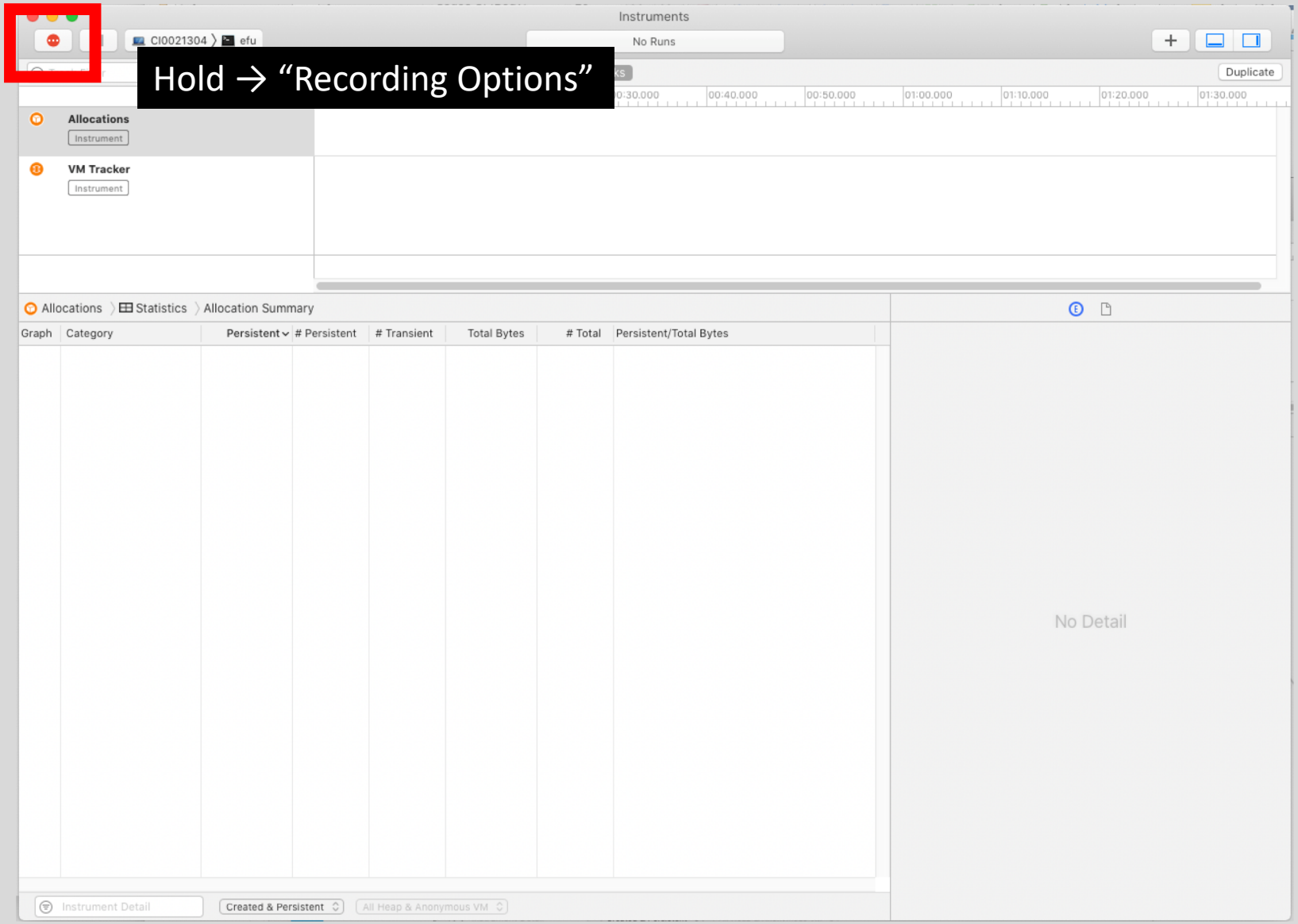


macOS Instruments: Transient allocations

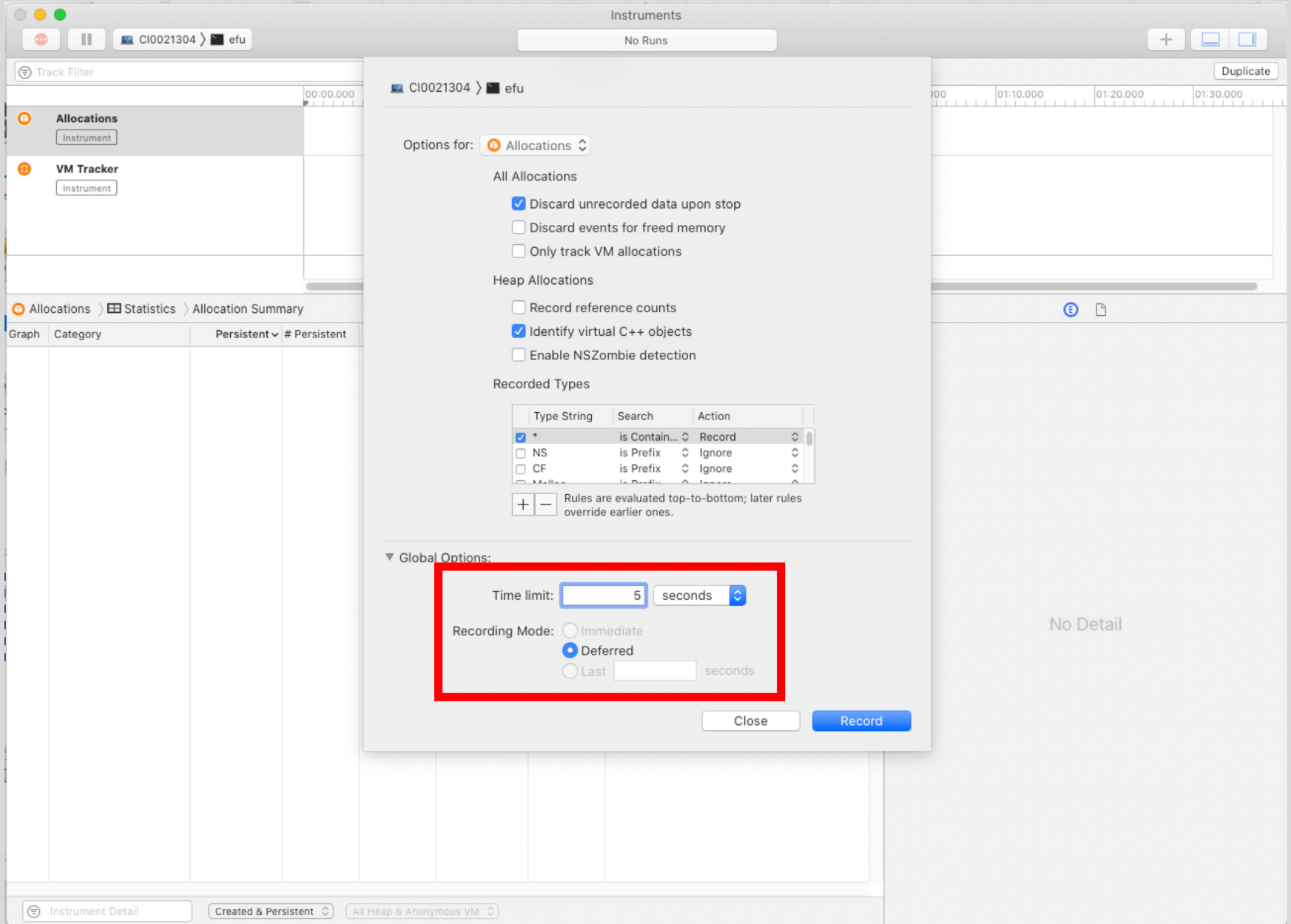


Can’t attach, must start target

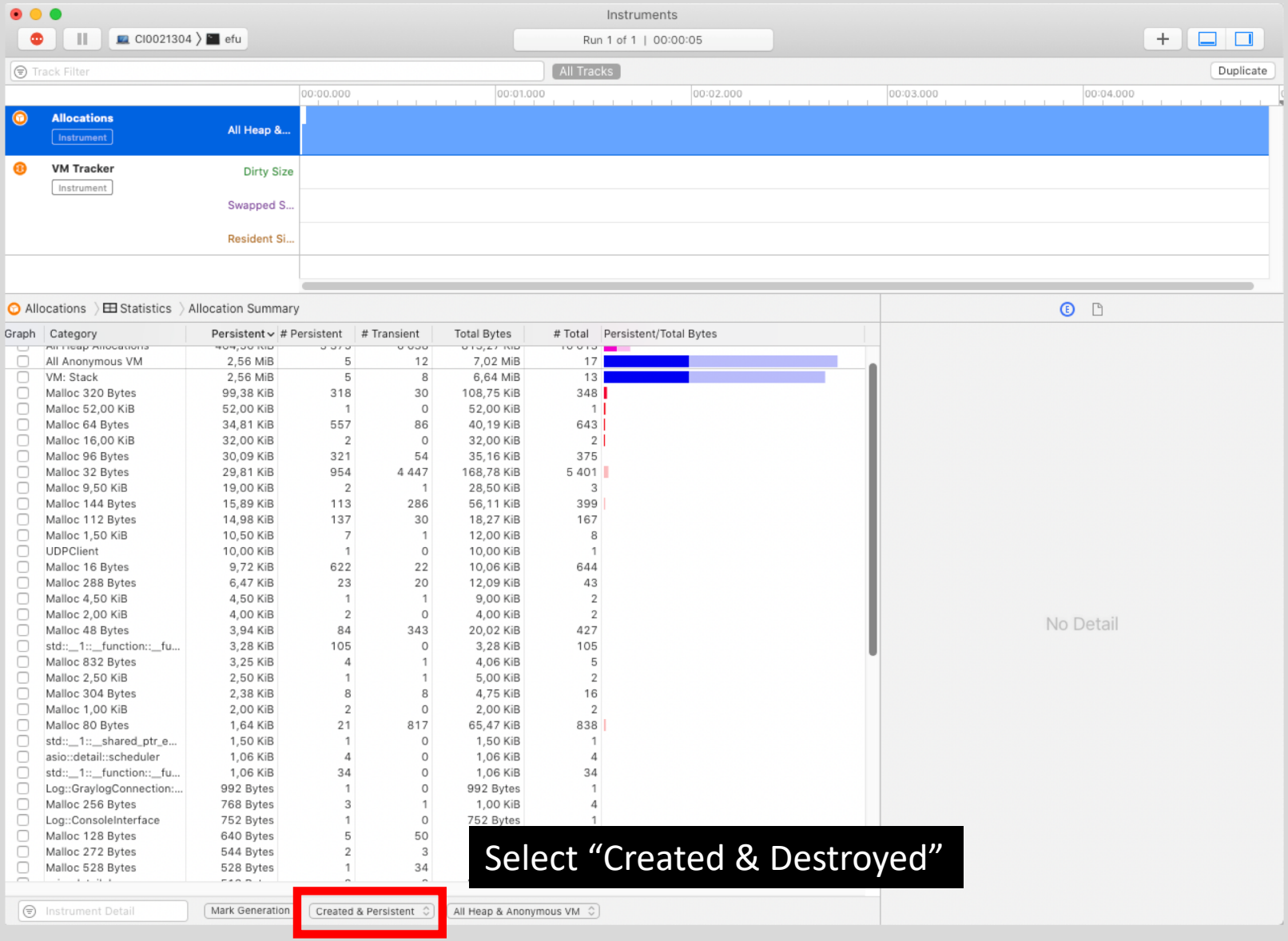
macOS Instruments: Transient allocations



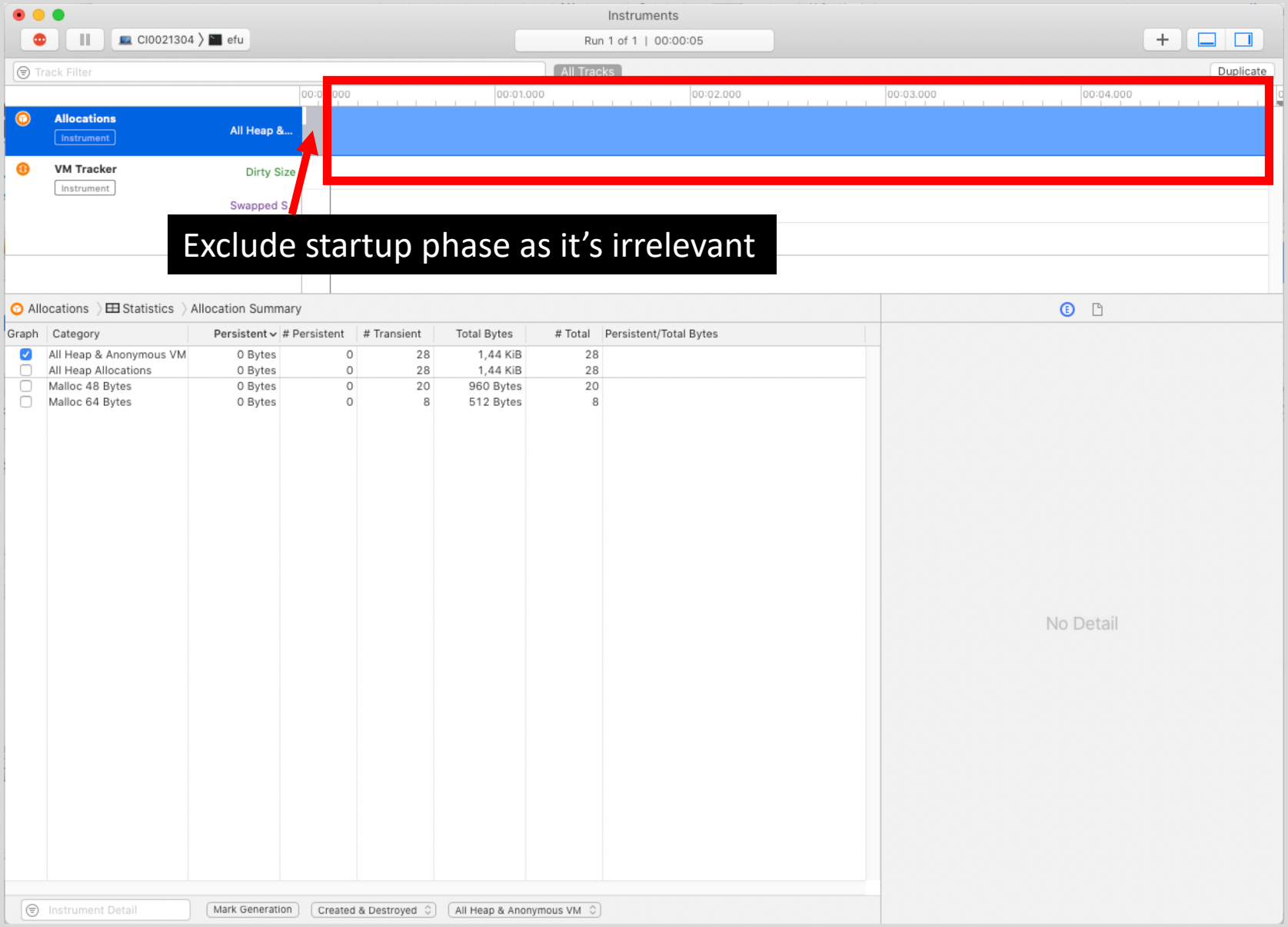
macOS Instruments: Transient allocations



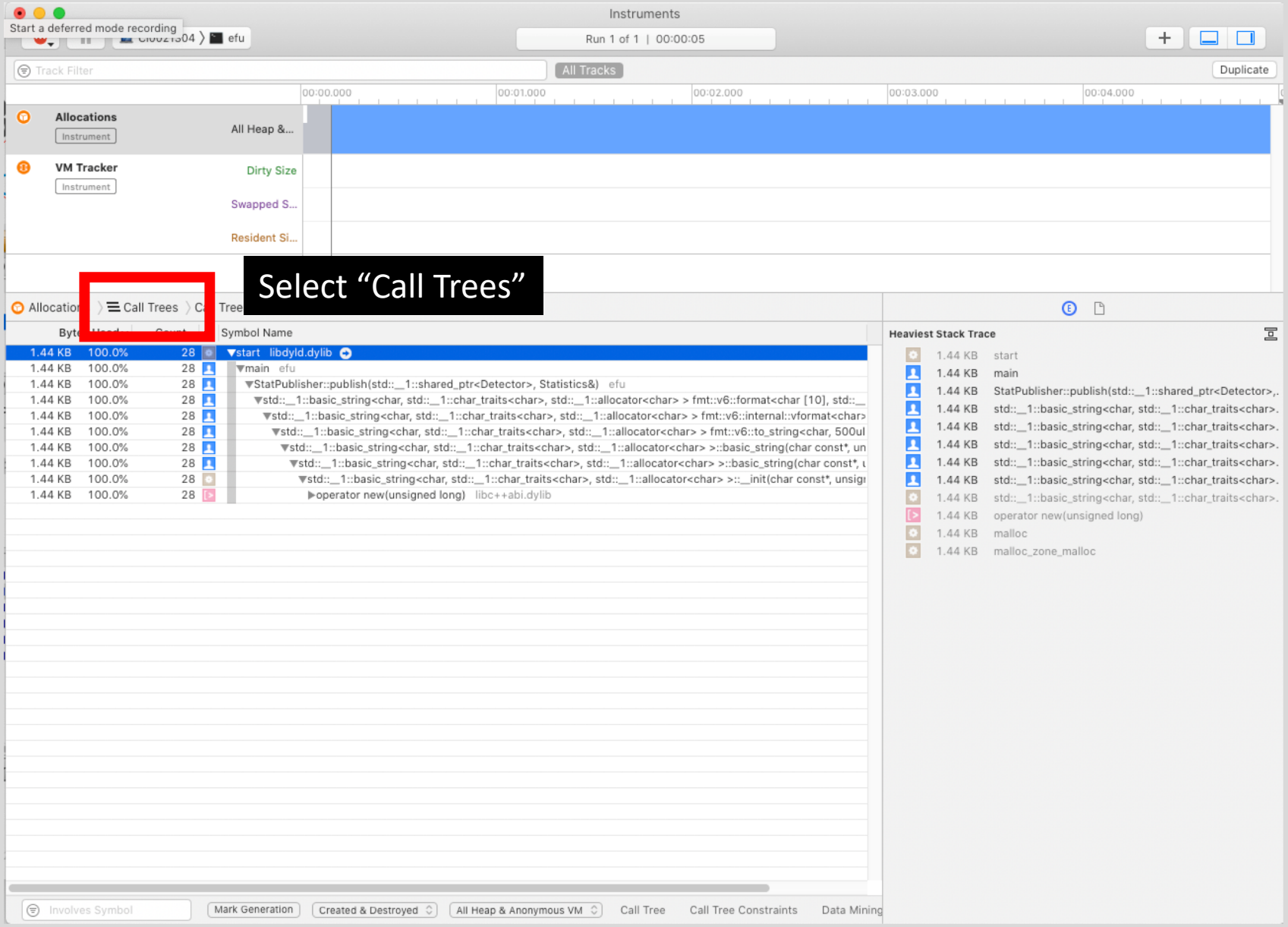
macOS Instruments: Transient allocations



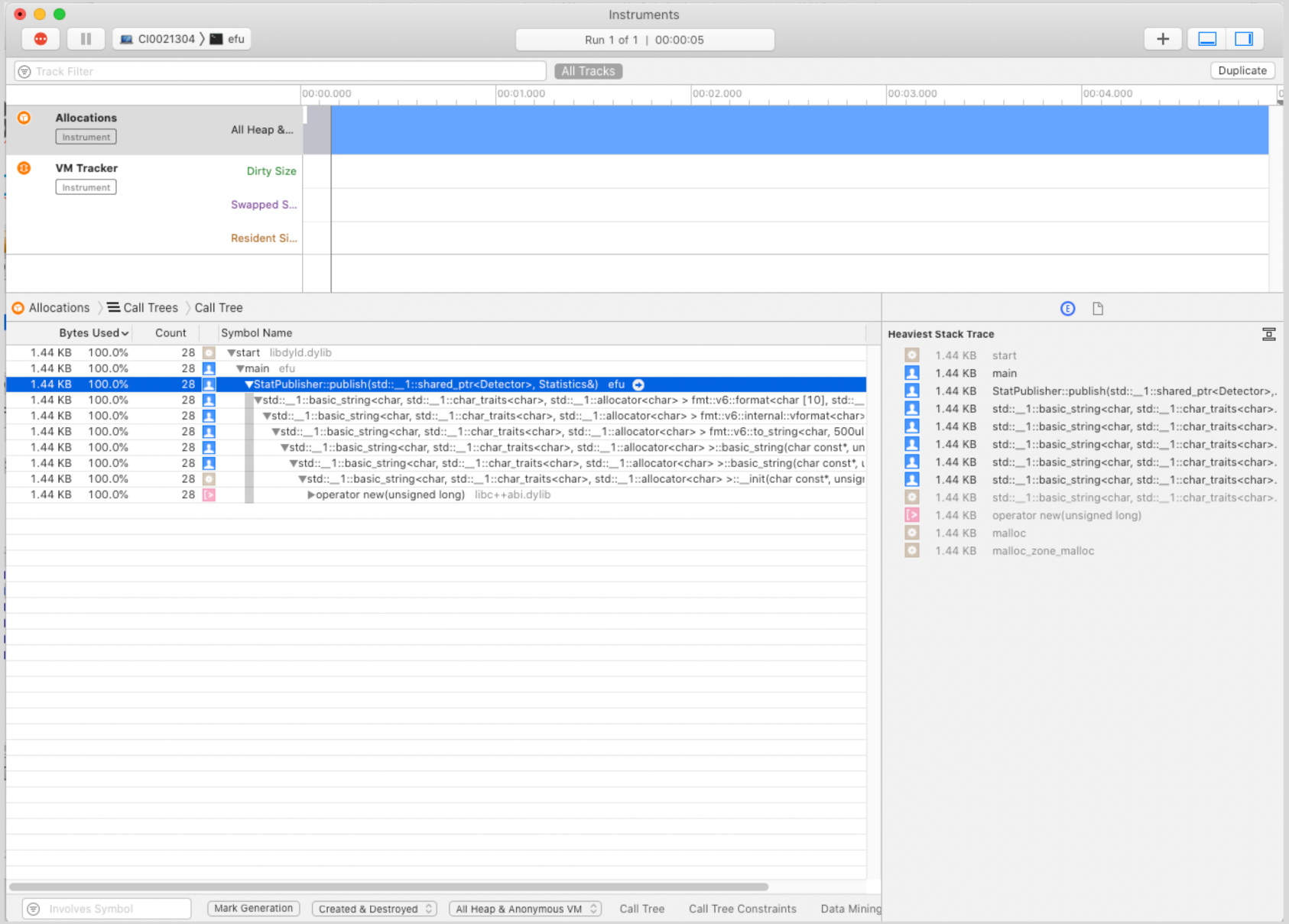
macOS Instruments: Transient allocations



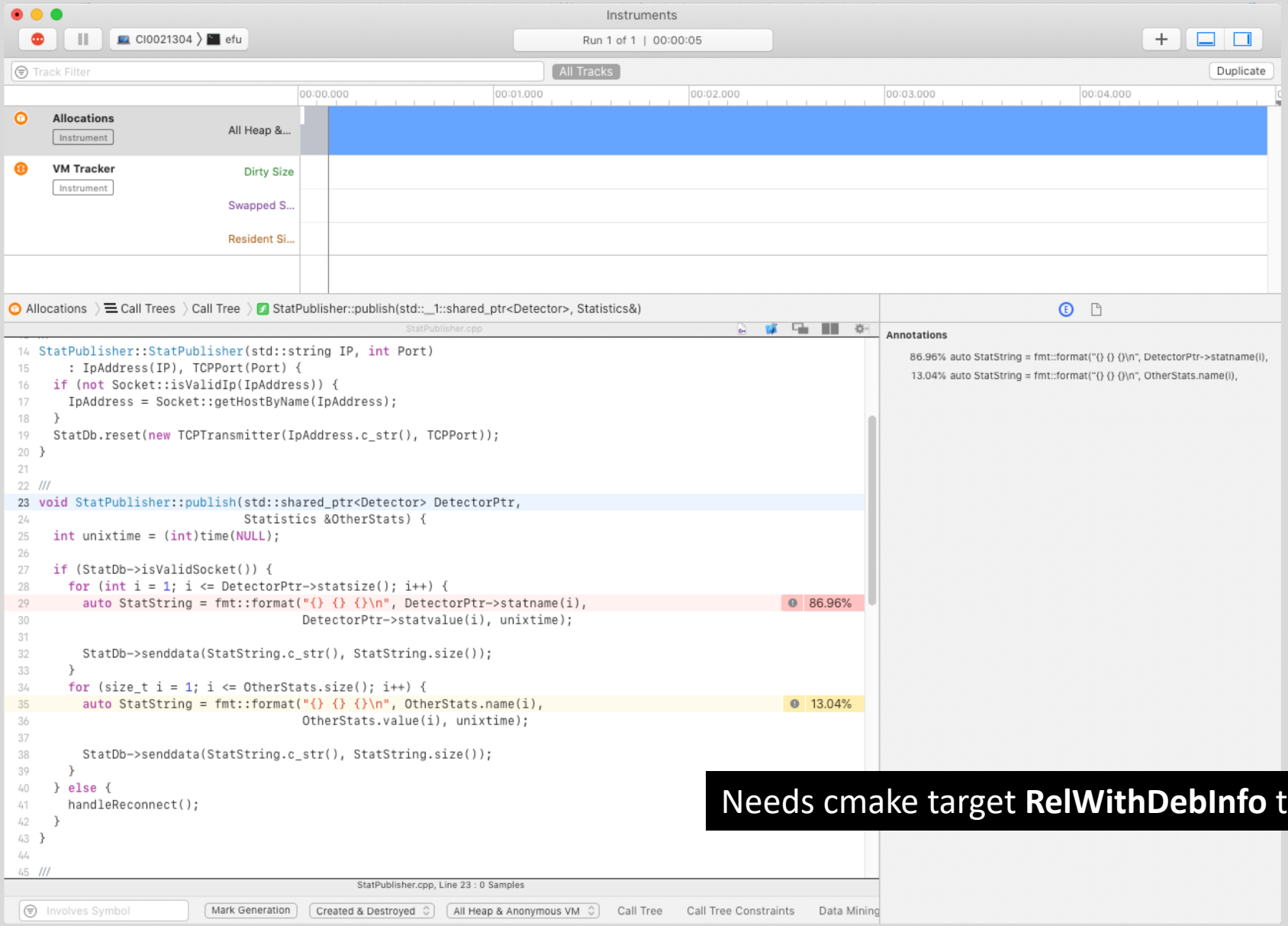
macOS Instruments: Transient allocations



macOS Instruments: Transient allocations



macOS Instruments: Transient allocations



Needs cmake target **RelWithDebInfo** to find source

macOS Instruments: Transient allocations

Conclusion

StatPublisher not likely a bottleneck, try fixed-size buffer for formatting

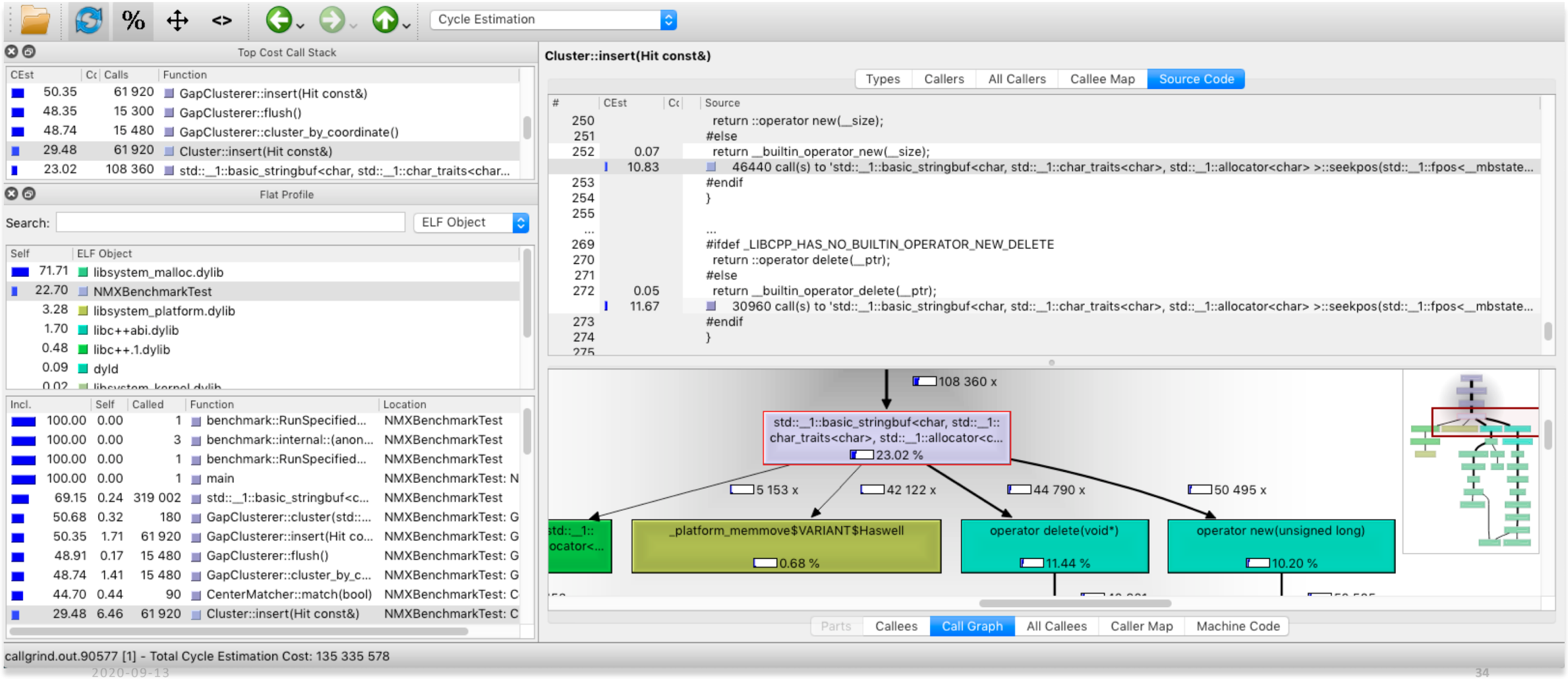
👍 Easy to map allocations to source code

3

Mention Callgrind + QCachegrind

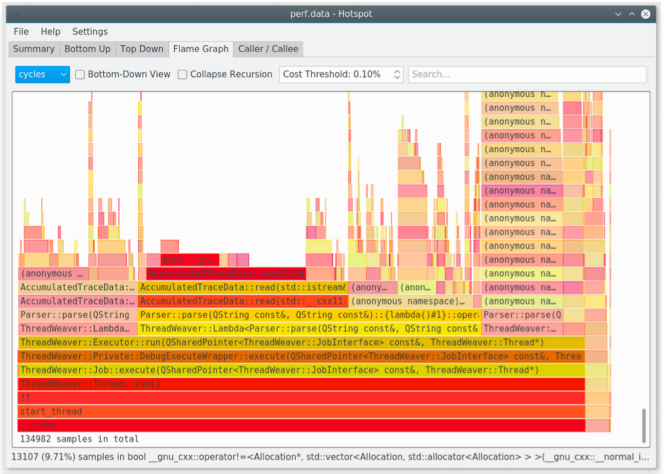
Mention Callgrind + QCachegrind

- Mac + Linux
- Full install guide & walkthrough [on Confluence]

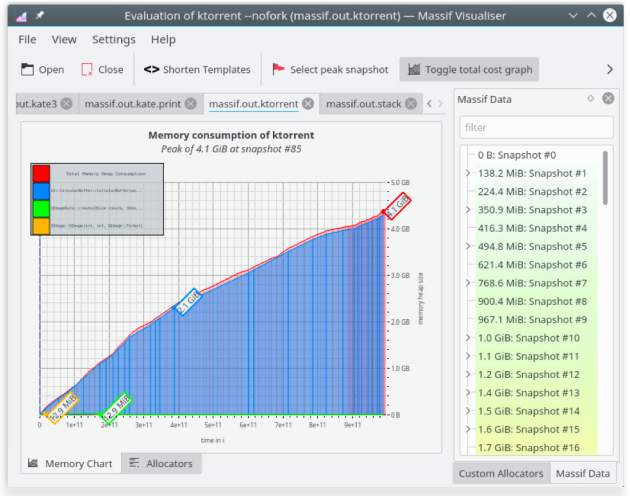


Mention Callgrind + QCachegrind

- Callgrind is Mac+Linux, try dedicated Linux tools:
- Hotspot (perf)



- Massif Visualizer (mem)



- Heaptrack (mem)

4

Understanding machine code using Ghidra

Understanding machine code using Ghidra

Open-source binary reverse engineering tool by NSA

Tries to generate source code from machine code

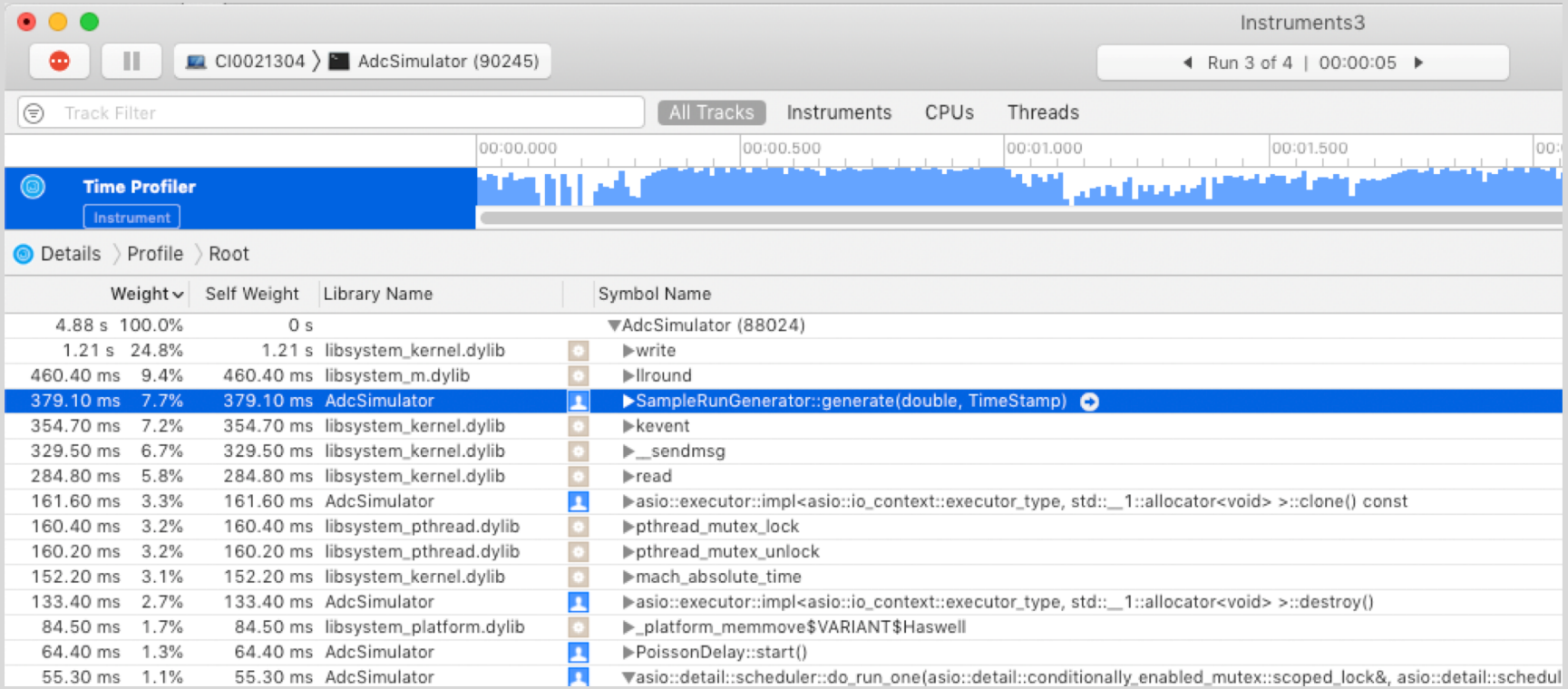
Can be installed using brew, instructions here (also needs to install Java)

Create a project and import binary.

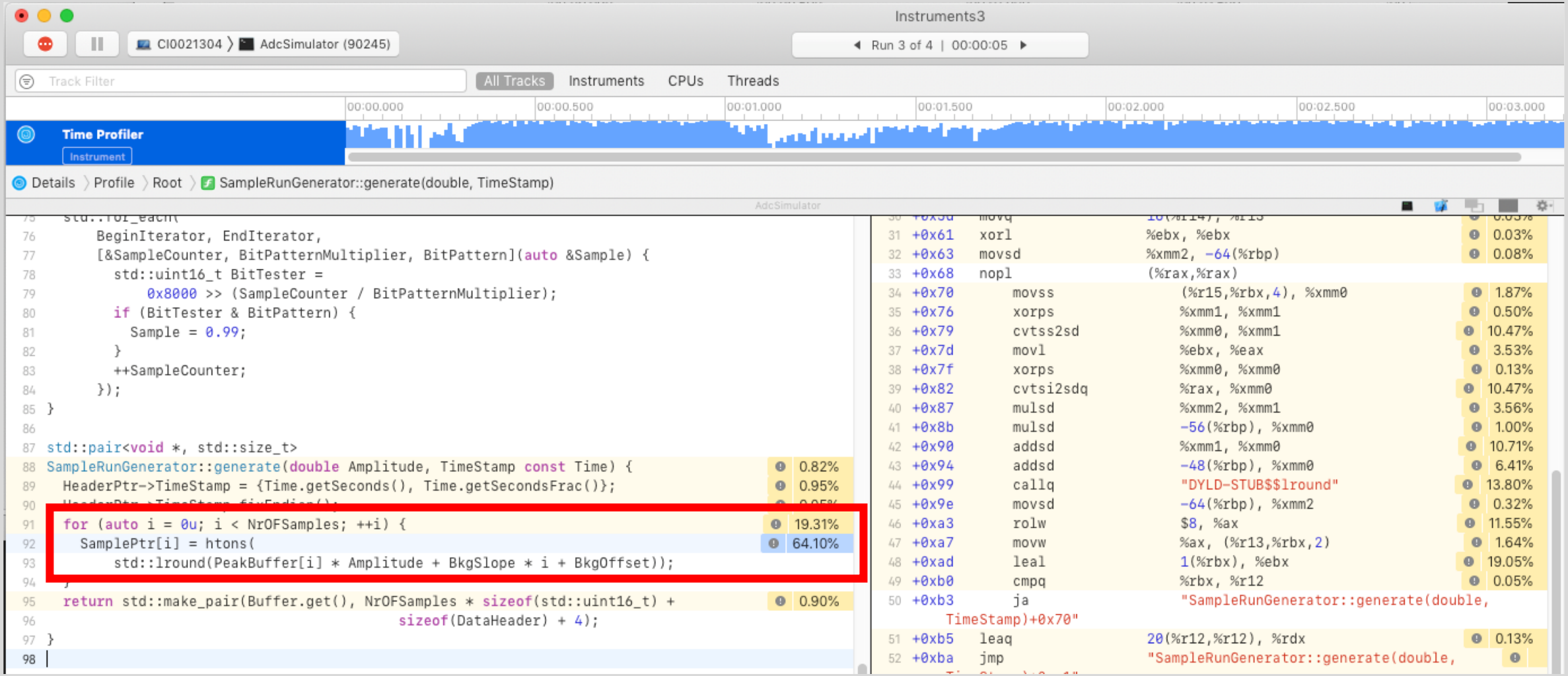
Names are mangled, but you can search using “*” instead of “::”, etc.

Case: generate() loop in AdcSimulator

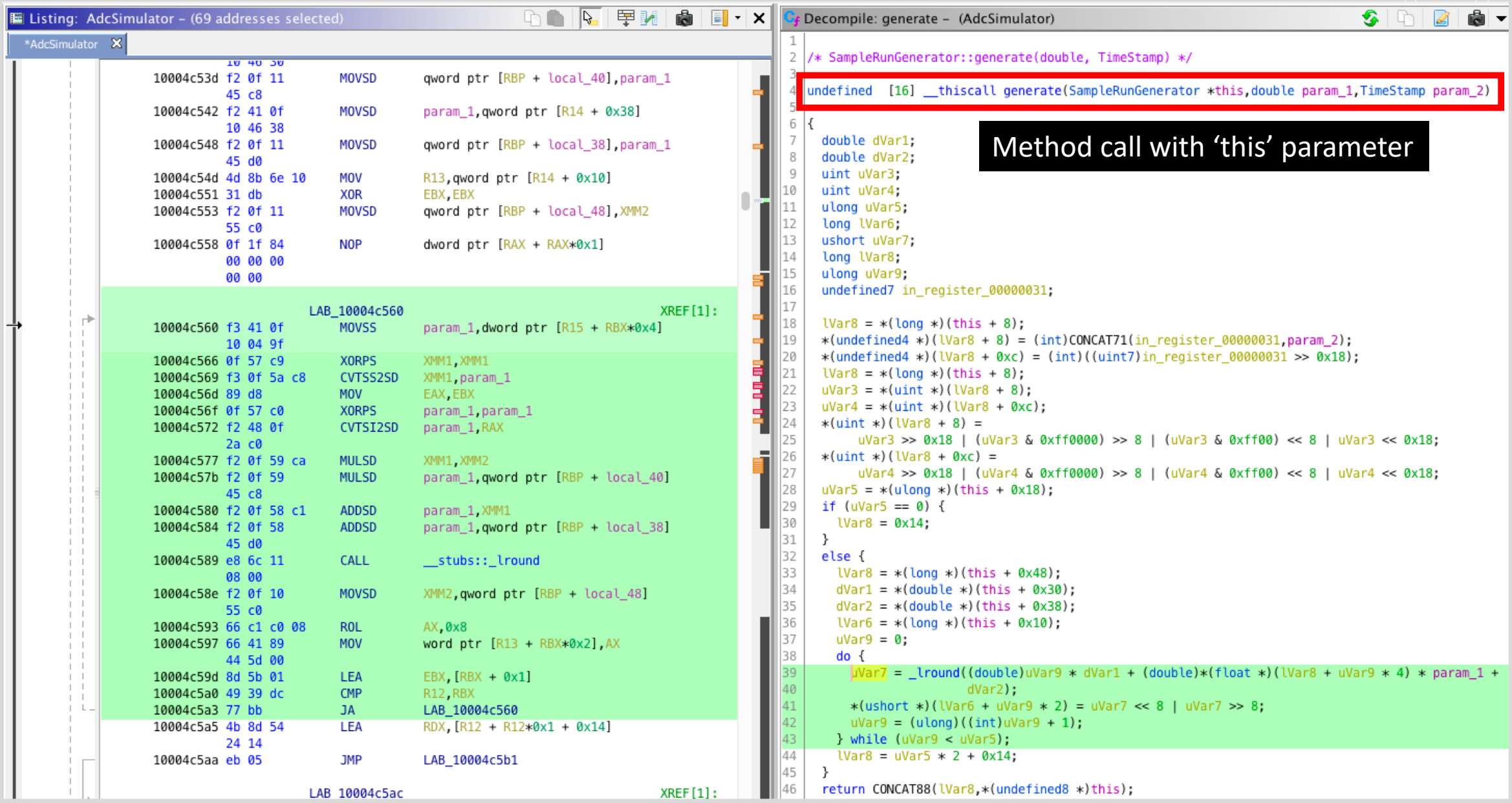
Understanding machine code using Ghidra



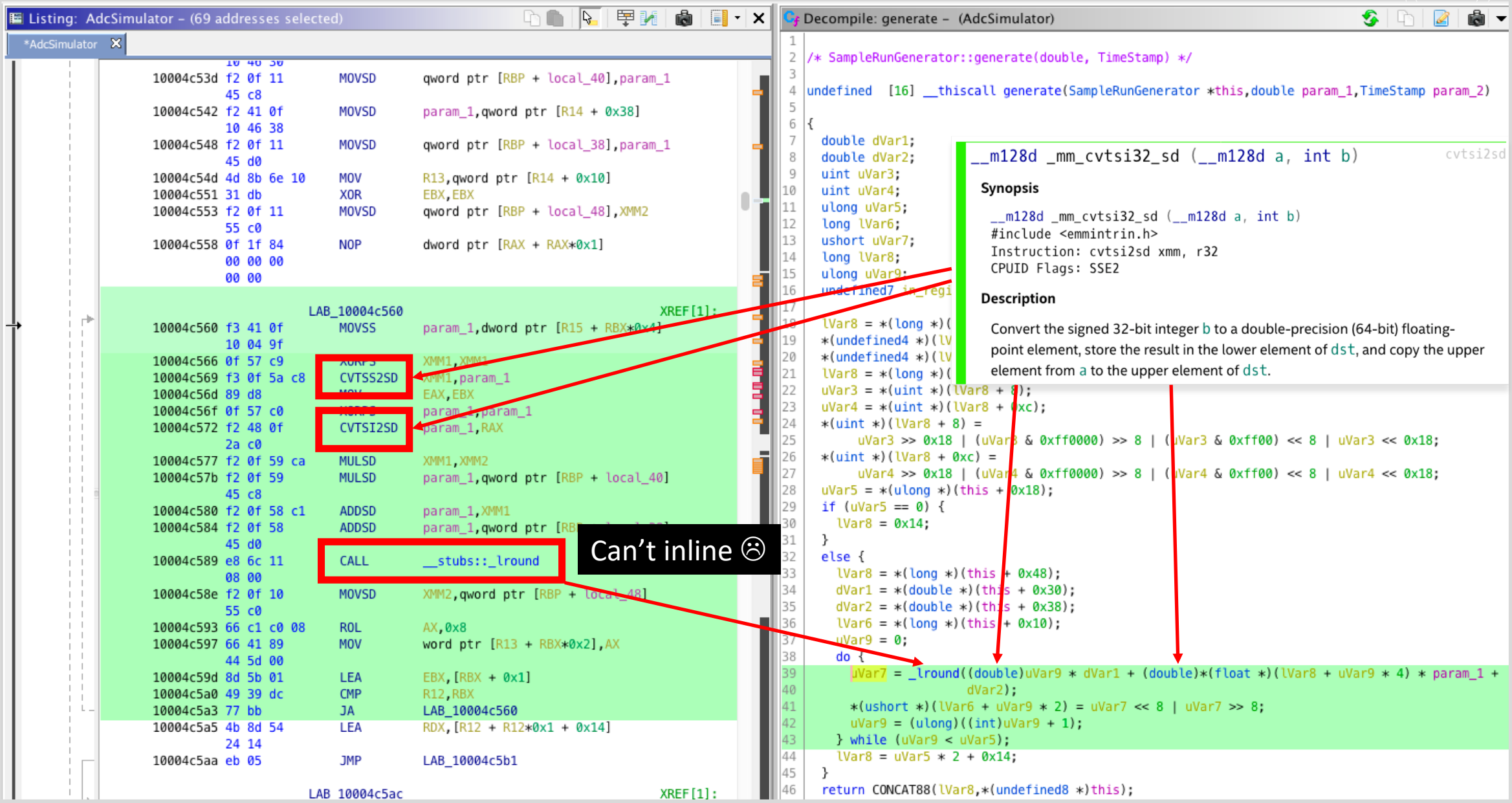
Understanding machine code using Ghidra



Understanding machine code using Ghidra



Understanding machine code using Ghidra



Understanding machine code using Ghidra

Conclusion

Let’s disallow implicit numeric conversion

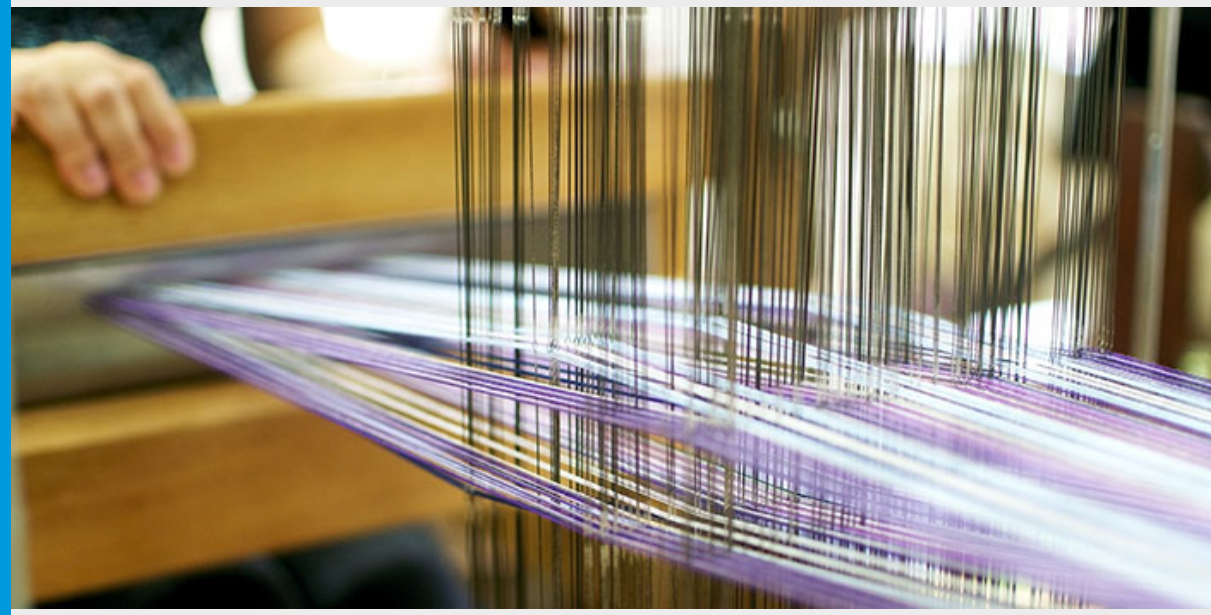
Can’t predict what functions can be inlined

Implies we can’t predict whether C++ compiler will vectorize code

Rewrote in ISPC to target 8 floats with AVX2 (fast enough)

5

Fast code with ISPC language + compiler



Source: Safan

Fast code with ISPC language + compiler

Explicit parallelism in an shader-like manner

Data-parallel, run N instances of the same function, on N sets of input arguments, using N SIMD lanes.

Paper (13 pages)

Example using 4 program instances

```
int f(int a, int b) {
  if (a < 0)
    a = 3;
  else
    a += b;
  return a;
}
```

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {

        if (a < 0)

            a = 3;


        else

            a += b;


    return a;
}
```

int a: [-2, -1, 0, 1]

int b: [1, 2, 3, 4]

int mask m: [_, _, _, _]

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {
    if (a < 0)
        a = 3;
    else
        a += b;
    return a;
}
```

int a: [-2, -1, 0, 1] int b: [1, 2, 3, 4] int mask m: [_, _, _, _]

m: [F, F, 0, 0] ← a: [-2, -1, 1, 2] < [0, 0, 0, 0]

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {
    if (a < 0)
        a = 3;
    else
        a += b;
    return a;
}
```

int a: [-2, -1, 0, 1] int b: [1, 2, 3, 4] int mask m: [_, _, _, _]

m: [F, F, 0, 0] ← a: [-2, -1, 1, 2] < [0, 0, 0, 0]

a: [-2, -1, 1, 2] & m: [F, F, 0, 0] = [3, 3, 3, 3] & m: [F, F, 0, 0]

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {
```

int a: [-2, -1, 0, 1] int b: [1, 2, 3, 4] int mask m: [_, _, _, _]

```
    if (a < 0)
```

m: [F, F, 0, 0] ← a: [-2, -1, 1, 2] < [0, 0, 0, 0]

```
        a = 3;
```

a: [-2, -1, 1, 2] & m: [F, F, 0, 0] = [3, 3, 3, 3] & m: [F, F, 0, 0]

a: [3, 3, 1, 2] ← a: [-2, -1, _, _] = [3, 3, 0, 0]

```
    else
        a += b;

    return a;
}
```

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {
```

int a: [-2, -1, 0, 1] int b: [1, 2, 3, 4] int mask m: [_, _, _, _]

```
    if (a < 0)
```

m: [F, F, 0, 0] ← a: [-2, -1, 1, 2] < [0, 0, 0, 0]

```
        a = 3;
```

a: [-2, -1, 1, 2] & m: [F, F, 0, 0] = [3, 3, 3, 3] & m: [F, F, 0, 0]

a: [3, 3, 1, 2] ← a: [-2, -1, _, _] = [3, 3, 0, 0]

```
    else
```

m: [0, 0, F, F] ← ~ m: [F, F, 0, 0]

```
        a += b;

    return a;
}
```

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {        int a: [-2, -1, 0, 1]      int b: [1, 2, 3, 4]      int mask m: [_, _, _, _]

    if (a < 0)               m: [F, F, 0, 0]   ← a: [-2, -1, 1, 2]   <   [0, 0, 0, 0]

        a = 3;                                     a: [-2, -1, 1, 2]   &  m: [F, F, 0, 0]   =  [3, 3, 3, 3]   &  m: [F, F, 0, 0]
                                                                          ↓                                          ↓
                             a: [3, 3, 1, 2]   ←                  a: [-2, -1, _, _]                =                [3, 3, 0, 0]

    else                     m: [0, 0, F, F]   ← ~ m: [F, F, 0, 0]

        a += b;                                    a: [3, 3, 1, 2]   &  m: [0, 0, F, F]   +=  b: [1, 2, 3, 4]   &  m: [0, 0, F, F]


    return a;
}
```

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {        int a: [-2, -1, 0, 1]      int b: [1, 2, 3, 4]      int mask m: [_, _, _, _]

    if (a < 0)               m: [F, F, 0, 0]   ← a: [-2, -1, 1, 2]   <   [0, 0, 0, 0]

        a = 3;                                     a: [-2, -1, 1, 2]   &  m: [F, F, 0, 0]   =  [3, 3, 3, 3]   &  m: [F, F, 0, 0]
                                                                       ↓                                        ↓
                             a: [3, 3, 1, 2]   ←                  a: [-2, -1, _, _]                =          [3, 3, 0, 0]

    else                     m: [0, 0, F, F]   ← ~ m: [F, F, 0, 0]

        a += b;                                    a: [3, 3, 1, 2]   &  m: [0, 0, F, F]   +=  b: [1, 2, 3, 4]   &  m: [0, 0, F, F]
                                                                       ↓                                        ↓
                             a: [3, 3, 3, 4]   ←                  a: [_, _, 1, 2]                  +=         [0, 0, 3, 4]

    return a;
}
```

Fast code with ISPC language + compiler

(Format) variable: [instance1, instance2, instance3, instance4]

```
int f(int a, int b) {        int a: [-2, -1, 0, 1]      int b: [1, 2, 3, 4]      int mask m: [_, _, _, _]

    if (a < 0)               m: [F, F, 0, 0]   ← a: [-2, -1, 1, 2]   <   [0, 0, 0, 0]

        a = 3;                                 a: [-2, -1, 1, 2]   &  m: [F, F, 0, 0]   =  [3, 3, 3, 3]   &  m: [F, F, 0, 0]
                                               ↓                                                            ↓
                             a: [3, 3, 1, 2]   ←        a: [-2, -1, _, _]                                =        [3, 3, 0, 0]

    else                     m: [0, 0, F, F]   ← ~  m: [F, F, 0, 0]

        a += b;                                a: [3, 3, 1, 2]   &  m: [0, 0, F, F]   +=  b: [1, 2, 3, 4]   &  m: [0, 0, F, F]
                                               ↓                                                            ↓
                             a: [3, 3, 3, 4]   ←        a: [_, _, 1, 2]                                  +=       [0, 0, 3, 4]

    return a;                return a: [3, 3, 3, 4]
}
```

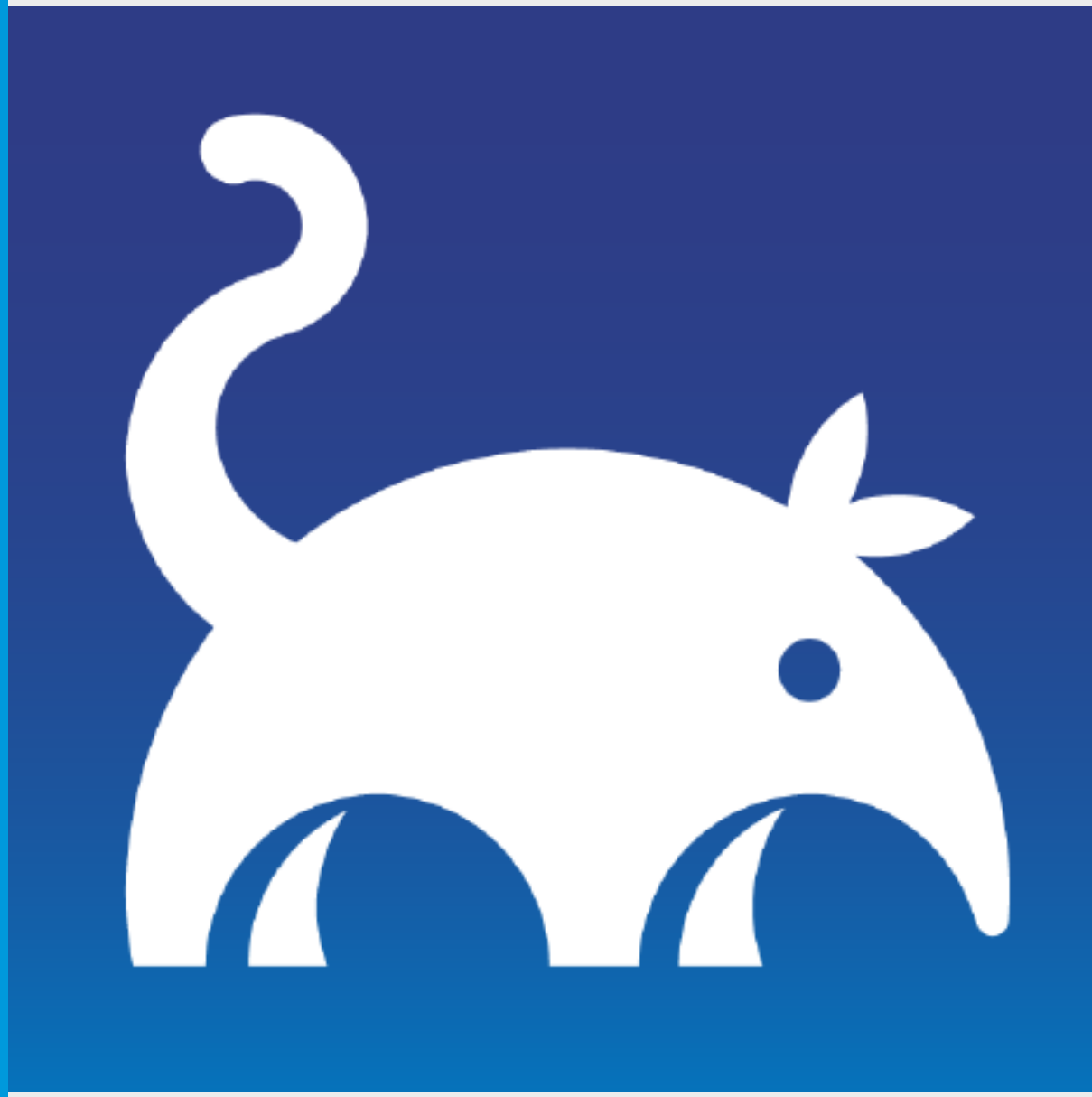
Fast code with ISPC language + compiler

Conclusion

- Example had 4 simultaneous instances, scales to 8, 16 .. N, GPU in future
- Useful for processing independent data “cells”
- Used for converting data when writing network packets (high speed)
- Hope to use to parse NMX packets, possibly process independent clusters?

6

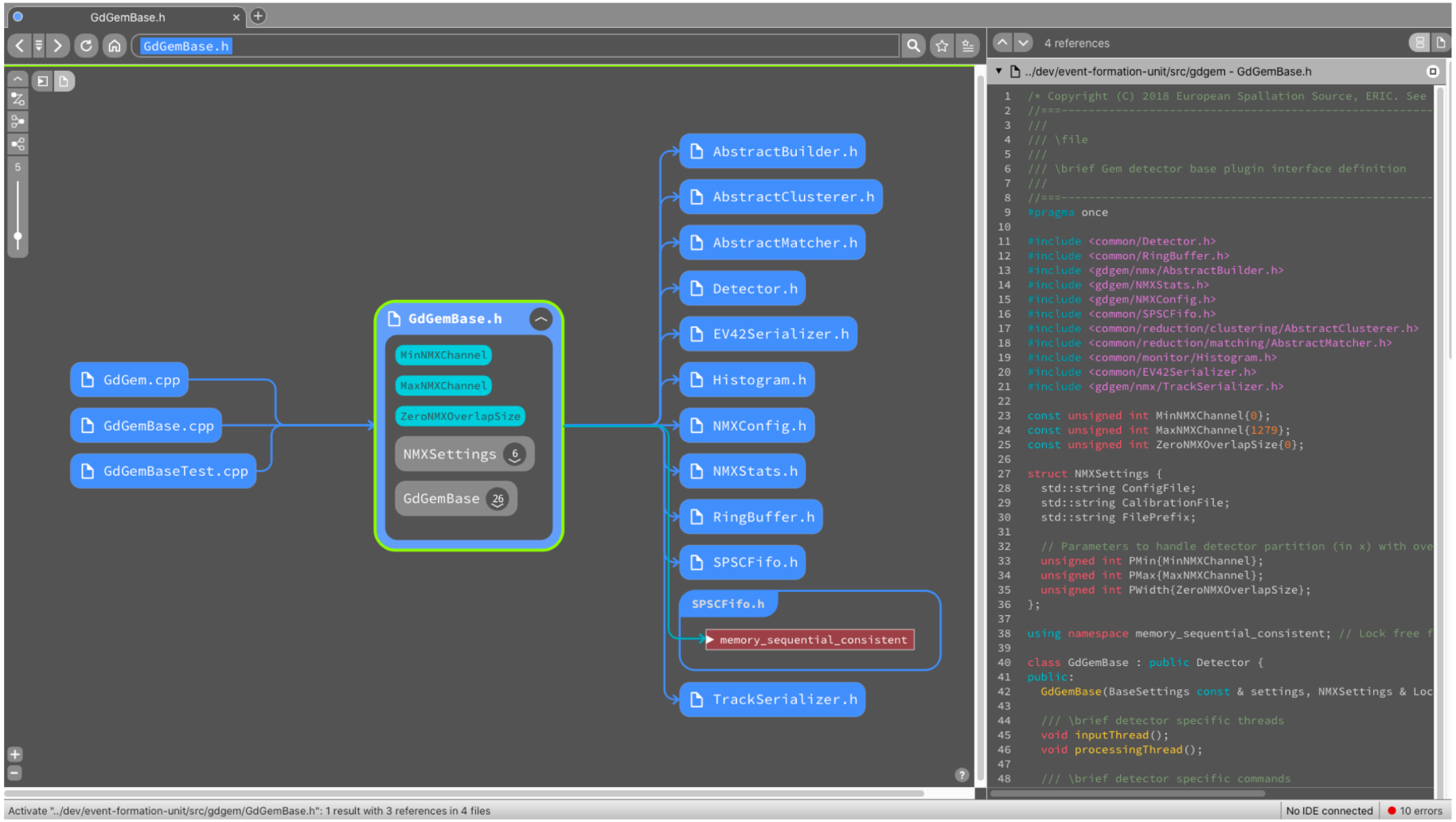
Viewing C++ projects with SourceTrail



Viewing C++ projects with SourceTrail

Open-source C++/Python code visualization tool [link]

Many ways of exploring symbols, implementations and references in a project



7

Viewing C++ compilation time

Source: XKCD: https://xkcd.com/303/

Viewing C++ compilation time profiles

Speed up Jenkins turnaround time

Guide to enable compile flags and run analysis

Resulting data files can be visualized by https://www.speedscope.app

Results aggregated by Clang Build Analyzer

Case: EFUArgs.cpp, 255 lines, 22 sec to compile

Viewing C++ compilation time profiles

Speed up Jenkins turnaround time

EFUArgs.cpp.json in https://www.speedscope.app

#include CLI.hpp
1.5 sec

Instanciate templates for CLI and fmt
7.0 sec

Optimize code for fmt
9.7 sec

Viewing C++ compilation time profiles

Clang Builder Analyzer, highest scoring entries

```
Files parse (compiler frontend):
                10466 ms: MultigridBase.cpp.o
Files codegen (compiler backend):
                18433 ms: udpgen_jalousie/generator.cpp.o
Templates instantiate:
                15338 ms: fmt::v6::internal::vformat<char>
Templates sets instantiate:
                171281 ms: std::__1::function<$>::function<$>
Functions compile:
                821 ms: CLI::App::_parse_arg(std::__1::vector<...
Function sets compile / optimize:
                115961 ms: void fmt::v6::internal::basic_writer<$>...
Expensive headers:
                56290 ms: include/CLI/CLI.hpp (included 40 times, avg 1407 ms), included via ...
```

Viewing C++ compilation time profiles

Conclusion

Improve turnaround time by optimizing build time, it takes 15 min now, what in 3 years? 10?

Look into pre-defining “extern template” to recycle non-inlined code.

Look into Piml-like ways of hiding private members

Measure possible speed gain by creating mock macro definitions

Finish presentation