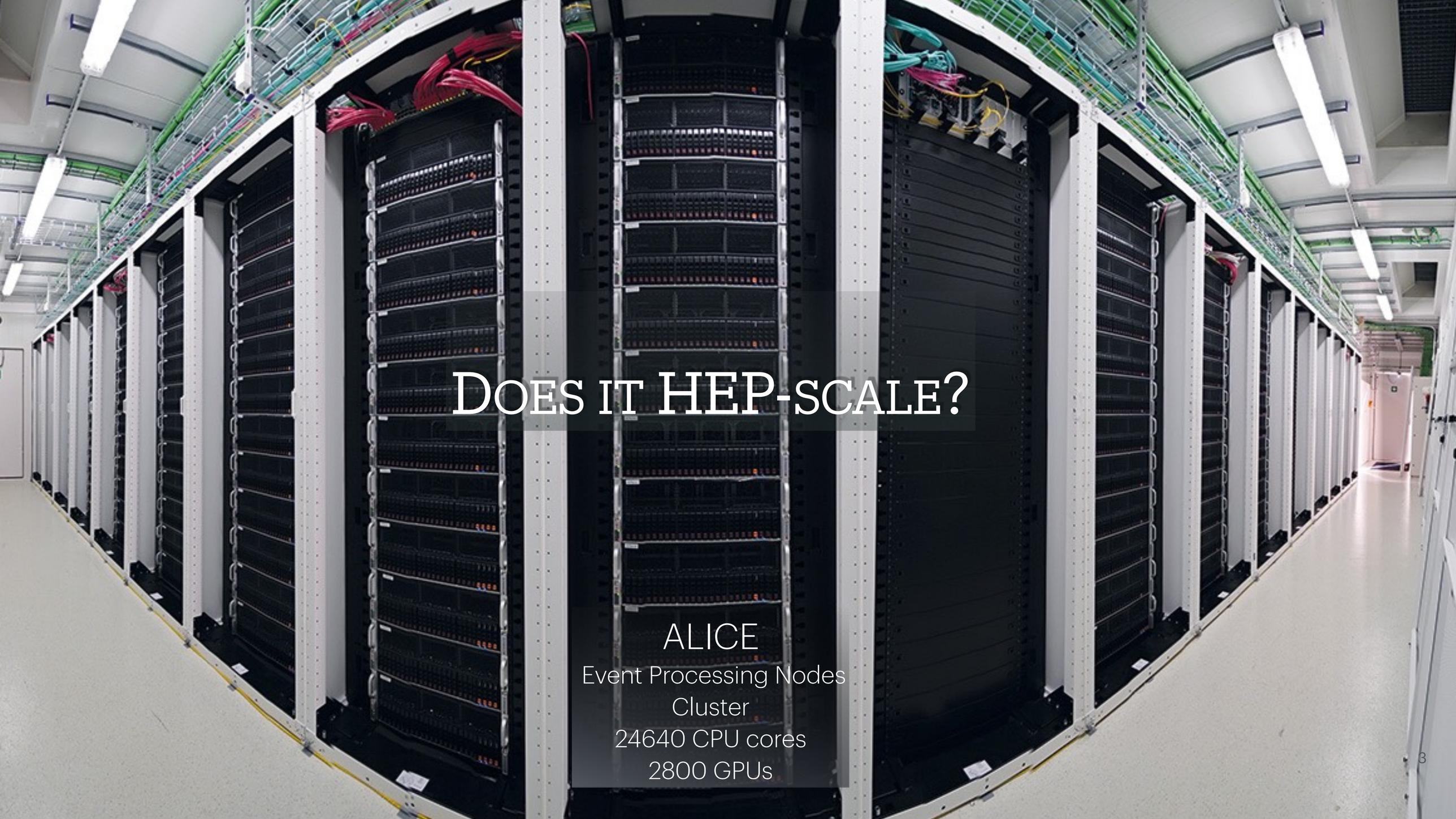
Benchmarking & Profiling

Your software is now working correctly on your laptop...



Does it HEP-scale?

ALICE

Event Processing Nodes

Cluster

24640 CPU cores

2800 GPUs

HEP Software outlook

High computing requirements: current and future generation experiments are very CPU / storage hungry. A few percentage of speed improvement might result in big savings in terms of resources and ultimately (experiment) money.

Heterogeneous environment: running on the grid, HEP software is faced with a plethora of computing hardware with different performance characteristics and operating environment. From TOP500 supercomputer to legacy 10 years old hardware in remote locations.

It's therefore important to understand the **effect of our design choices** on the performance.

What is the time complexity of the following assignment?

```
a[0] = 1;
```

O(1)
(std::vector<int>)

```
a[0] = 1;
```

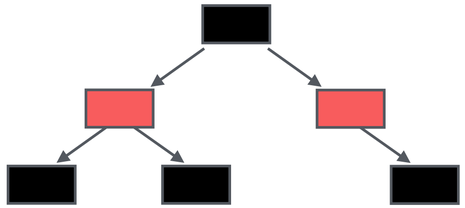
O(1)

(std::vector<int>)

```
a[0] = 1;
```

O(log_2N)

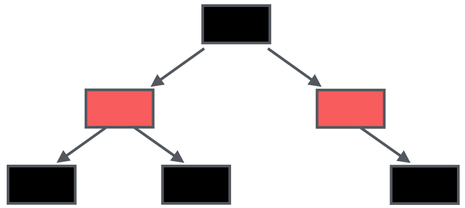
(std::map<int,int>)



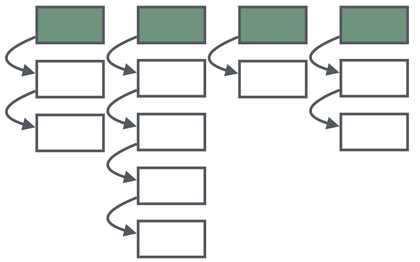
O(1)
(std::vector<int>)

```
a[0] = 1;
```

O(log_2N)
(std::map<int,int>)



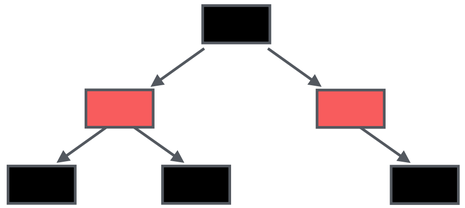
O(1)
(std::unordered_map<int,int>)



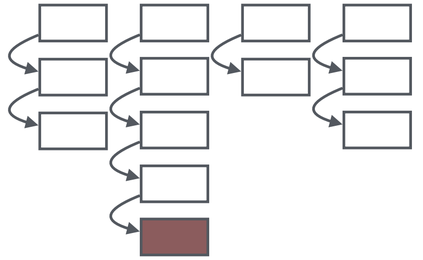
O(1)
(std::vector<int>)

```
a[0] = 1;
```

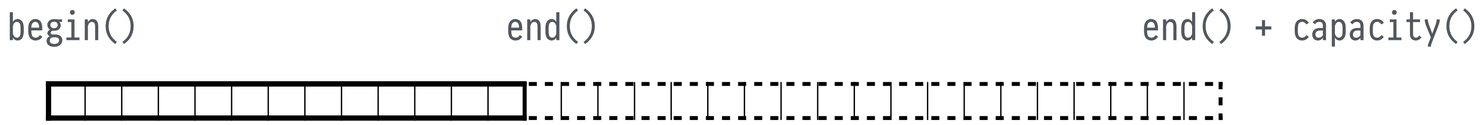
O(log_2N)
(std::map<int,int>)



O(N) (worst case scenario)
(std::unordered_map<int,int>)

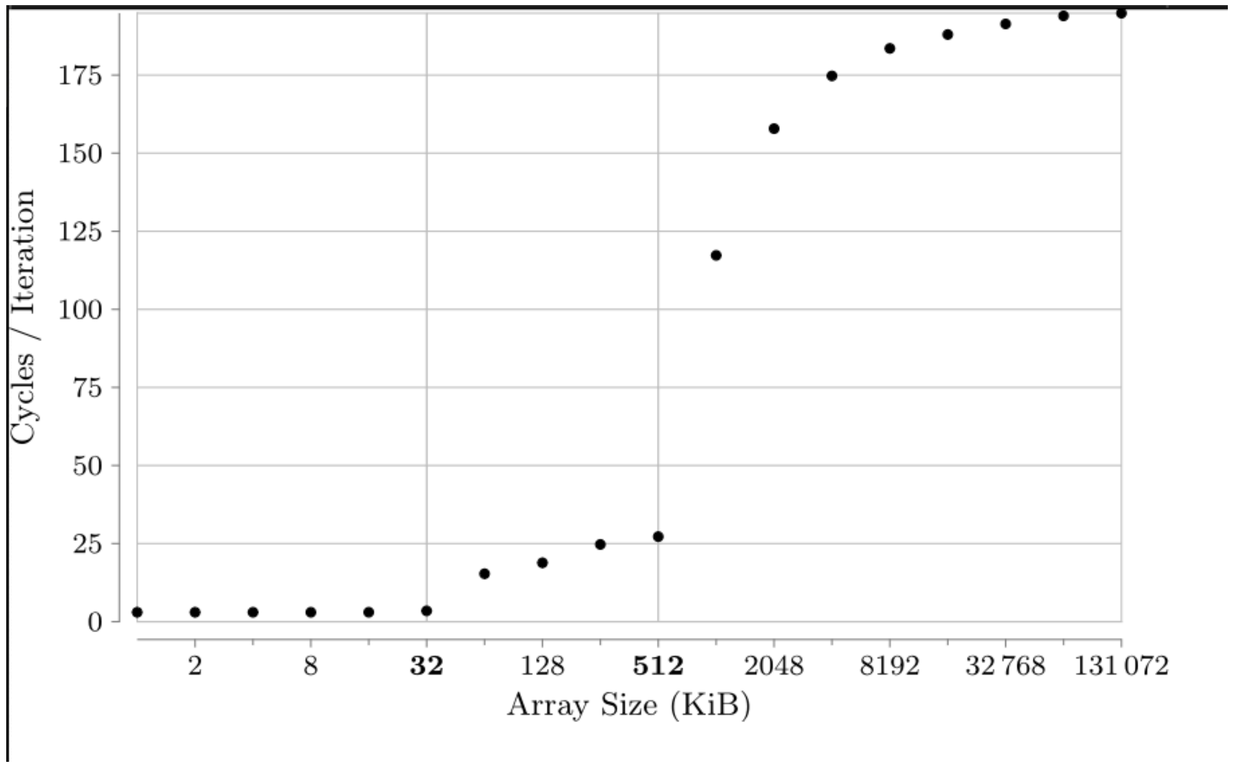


O(1) (std::vector<int>)

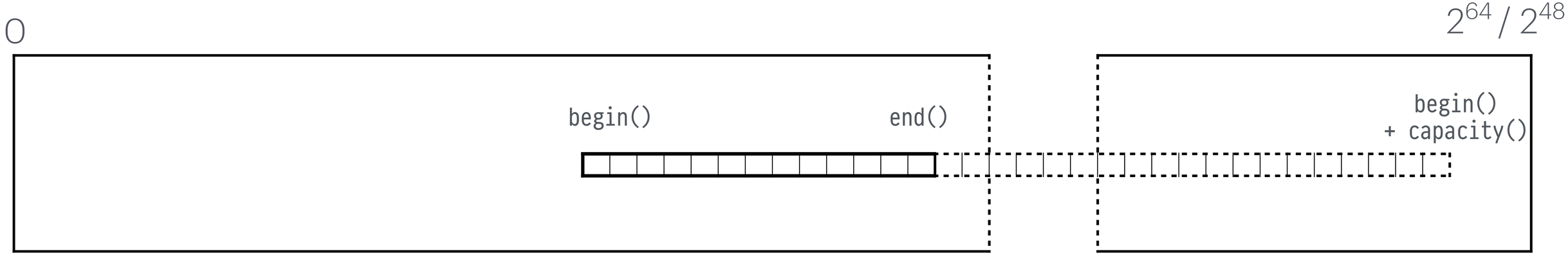


Access patterns matter as well!

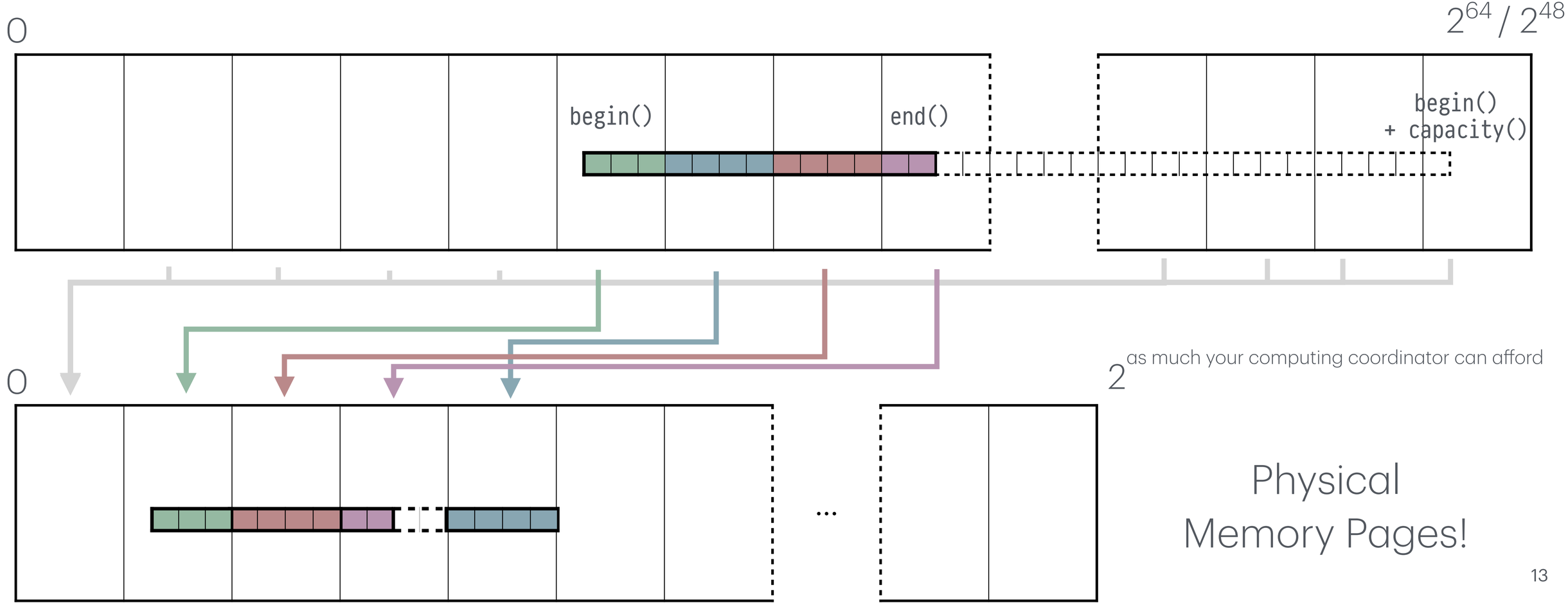
```
#define N 100000000  // 100 million

struct elem {
   struct elem *next;
} array[SIZE];

int main() {
   for (size_t i = 0; i < SIZE - 1; ++i) array[i].next = &array[i + 1];
   array[SIZE - 1].next = array;
   // Fisher-Yates shuffle the array.
   for (size_t i = 0; i < SIZE - 1; ++i) {
      size_t j = i + rand() % (SIZE - i);  // j is in [i, SIZE).
      struct elem temp = array[i];  // Swap array[i] and array[j].
      array[i] = array[j];
      array[j] = temp;
   }
#ifndef BASELINE
   int64_t dummy = 0;
   struct elem *i = array;
   for (size_t n = 0; n < N; ++n) {
      dummy += (int64_t)i;
      i = i->next;
   }
   printf("%d\n", dummy);
#endif
}
```

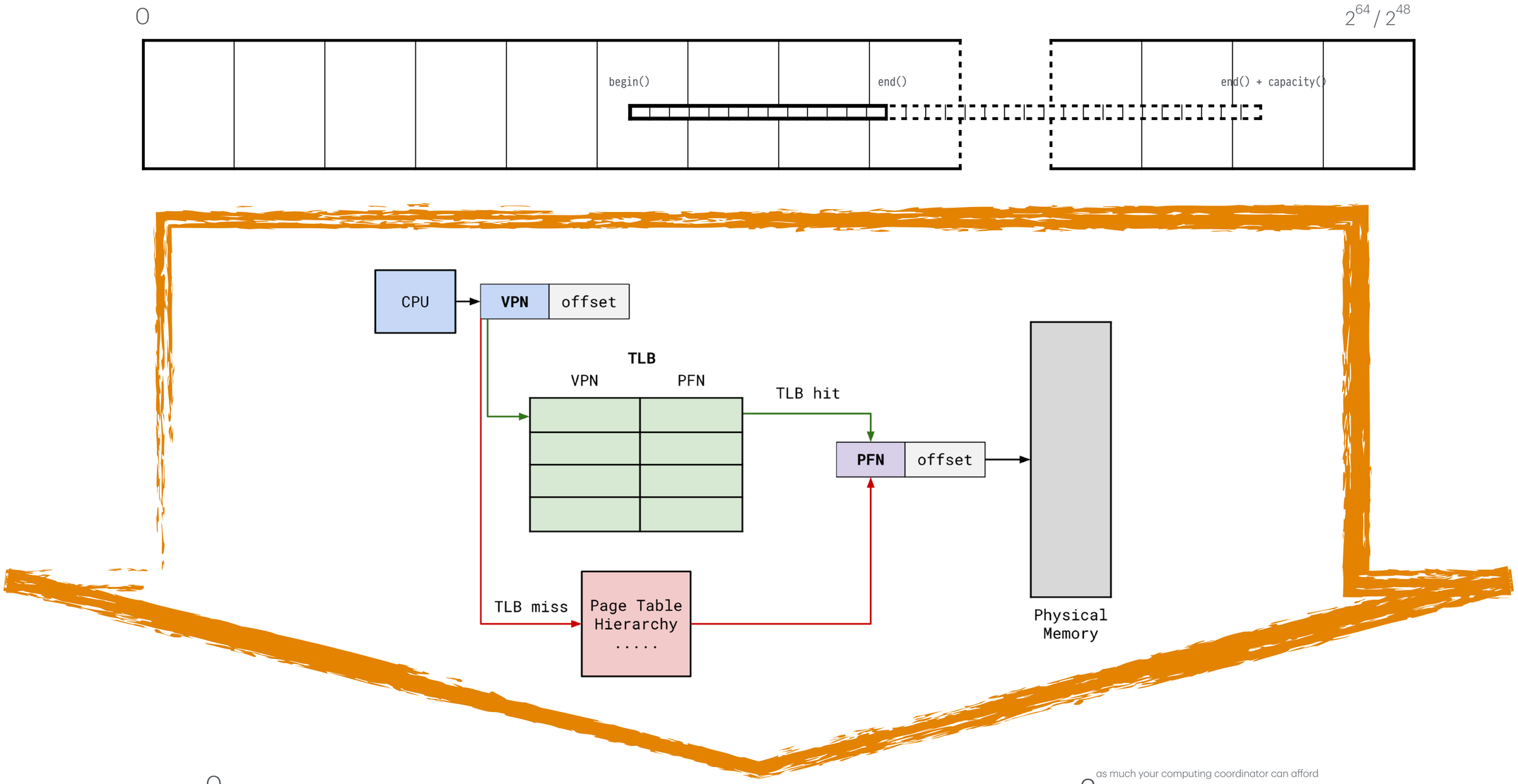


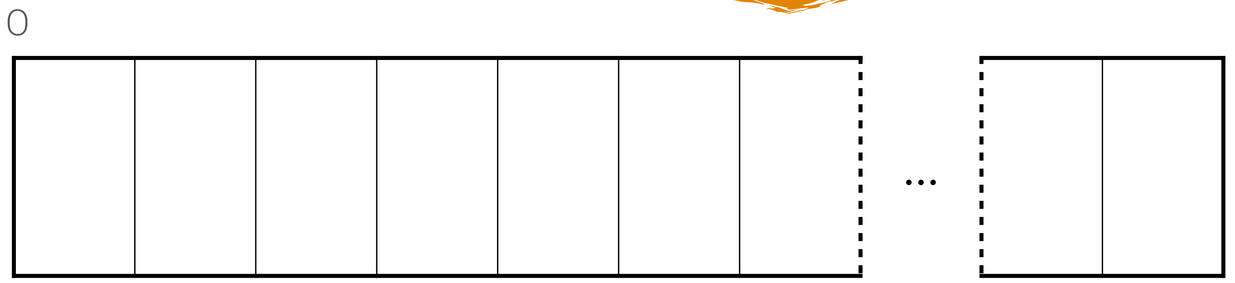
Flat Address Space



Flat **Virtual** Address Space







Physical Memory Pages!

Cache hierarchy

Caches always work in terms of **cache lines**: copies from / to the cache always happen in terms of cache-line size aligned chunks

Some processor have **multiple cores sharing a L3** cache

Some (e.g. Apple M1) have a **very fast memory bus** and no L3 cache


Programming
languages are just
an abstraction of
what happens inside
our computers

There are three key parts to improve the performance of your software:

1. measure
2. measure
3. measure

Benchmarking

$$\Delta t = t_{end} - t_{start}$$

Trivial benchmarking: time command

```
> cat busy-loop.cc
#include <cmath>

#define ITERATIONS 10000000000

int main(int, char**) {
  for (size_t i = 1; i <= ITERATIONS; ++i) {
    log(1);
  }
  return 0;
}
```

Trivial benchmarking: time command

```
> c++ -O2 busy-loop.cc -o busy-loop
> time ./busy-loop
./busy-loop  0.00s user 0.00s system 40% cpu 0.014 total
```

Trivial benchmarking: time command

```
> c++ -O2 busy-loop.cc -o busy-loop
> time ./busy-loop
./busy-loop  0.00s user 0.00s system 40% cpu 0.014 total
```

Is it reasonable?

Trivial benchmarking: time command

```
> cat busy-loop.cc
#include <cmath>

#define ITERATIONS 10000000000

int main(int, char**) {
  for (size_t i = 1; i <= ITERATIONS; ++i) {
    log(1);
  }
  return 0;
}
```

Back of the envelope calculation:

10^{10} iterations @ 5GHz* → 2s

In reality, our CPUs do not get to 5GHz and `log` takes more than one cycle to compute.

What is happening?

Let's have a look inside

```
> objdump -d busy-loop

busy-loop:      file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000100003fa0 <_main>:
100003fa0: 52800000     mov     w0, #0
100003fa4: d65f03c0     ret
```

The `objdump` command can be used to inspect the contents in your executable.

Here, all the program does is to set the register `w0` to the value `0` and return to the shell.

The **compiler was smart enough to optimise everything out**!

Trivial benchmarking: time command

```
> objdump -d busy-loop

busy-loop:      file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000100003fa0 <_main>:
100003fa0: 52800000     mov     w0, #0
100003fa4: d65f03c0     ret
```

The `objdump` command can be used to inspect the contents in your executable.

In this particular case all the program does is to set the register `w0` to the value `0` and return to the shell.

The compiler was smart enough to optimise everything out!

Trivial benchmarking: time command

```
> cat busy-loop.cc
#include <cmath>

#define ITERATIONS 10000000000

int main(int, char**) {
  for (size_t i = 1; i <= ITERATIONS; ++i) {
    log(1);
  }
  return 0;
}
```

The objdump command can be used to inspect the contents in your executable.

In this particular case all the program does is to set the register `w0` to the value `0` and return to the shell.

The compiler was smart enough to optimise dummy code out!

Trivial benchmarking: time command

```
int main(int, char**) {
  float r = -1;
  for (size_t i = 1; i <= ITERATIONS; ++i) {
    r = log(1);
  }
  return r;
}
```

```
0000000100003fa0 <_main>:
100003fa0: 52800000     mov     w0, #0
100003fa4: d65f03c0     ret
```

Adding a dependency of the result value on the log does not help either!

Once again the compiler is able to move invariants out of the loop...

Trivial benchmarking: time command

```
int main(int, char**) {
  float r = -1;
  for (size_t i = 1; i < ITERATIONS; ++i) {
    r = log(i);
  }
  return r;
}
```

```
0000000100003fa0 <_main>:
100003fa0: 52800120     mov     w0, #9
100003fa4: d65f03c0     ret
```

Adding a dependency of the result value on the log does not help either!

Once again the compiler is able to move invariants out of the loop...

...even adding a dependency on the loop variable!

Trivial benchmarking: time command

```
int main(int, char**) {
  float r = 0;
  for (size_t i = 1; i <= ITERATIONS; ++i) {
    r += log(i);
  }
  return r;
}
```

```
> c++ -O2 busy-loop.cc -o busy-loop
> time ./busy-loop

real    0m31.476s
user    0m31.253s
sys     0m0.008s
```

A dependency on the actual loop is needed to avoid optimisations.

```
0000000100003f44 <_main>:
100003f44: 6dbd23e9     stp     d9, d8, [sp, #-48]!
100003f48: a9014ff4     stp     x20, x19, [sp, #16]
100003f4c: a9027bfd     stp     x29, x30, [sp, #32]
100003f50: 910083fd     add     x29, sp, #32
100003f54: d2800013     mov     x19, #0
100003f58: 2f00e408     movi    d8, #0000000000000000
100003f5c: d29c8014     mov     x20, #58368
100003f60: f2aa8174     movk    x20, #21515, lsl #16
100003f64: f2c00054     movk    x20, #2, lsl #32
100003f68: 91000673     add     x19, x19, #1
100003f6c: 9e630260     ucvtf   d0, x19
100003f70: 9400000b     bl      0x100003f9c <_log+0x100003f9c>
100003f74: 1e22c101     fcvt    d1, s8
100003f78: 1e612800     fadd    d0, d0, d1
100003f7c: 1e624008     fcvt    s8, d0
100003f80: eb14027f     cmp     x19, x20
100003f84: 54ffff21     b.ne    0x100003f68 <_main+0x24>
100003f88: 1e380100     fcvtzs  w0, s8
100003f8c: a9427bfd     ldp     x29, x30, [sp, #32]
100003f90: a9414ff4     ldp     x20, x19, [sp, #16]
100003f94: 6cc323e9     ldp     d9, d8, [sp], #48
100003f98: d65f03c0     ret
```

Understand your code!

Compilers are smart and can do a whole lot of optimisation for you!

Compilers are also very strict and can produce very naive / slow code because the standard / compiler flags (`-fno-fast-math`!) forces them to do so!

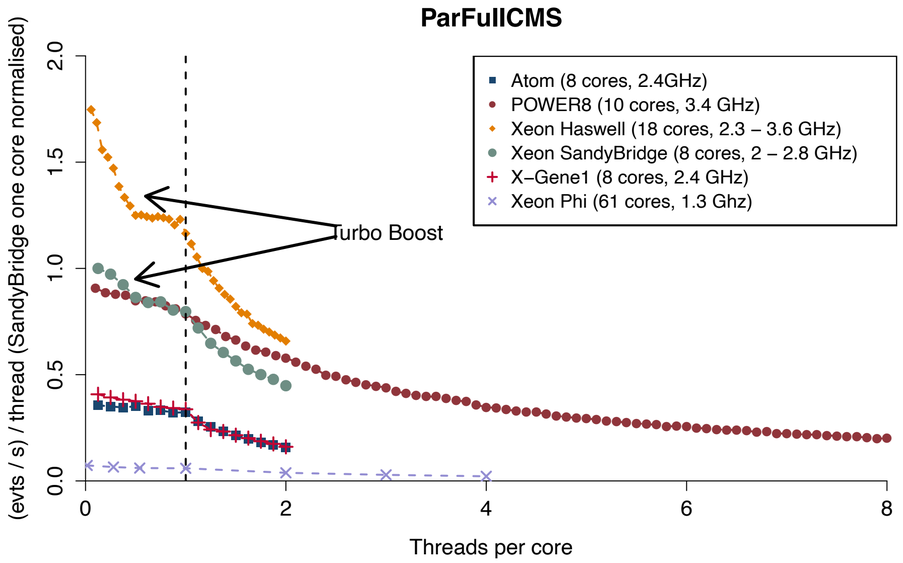
STL containers are very powerful, yet they have each their own use case. Despite the similar APIs, they have very different performance implications!

Always make sure you are measuring a realistic example and **model your expectations before you run a benchmark, not after!**

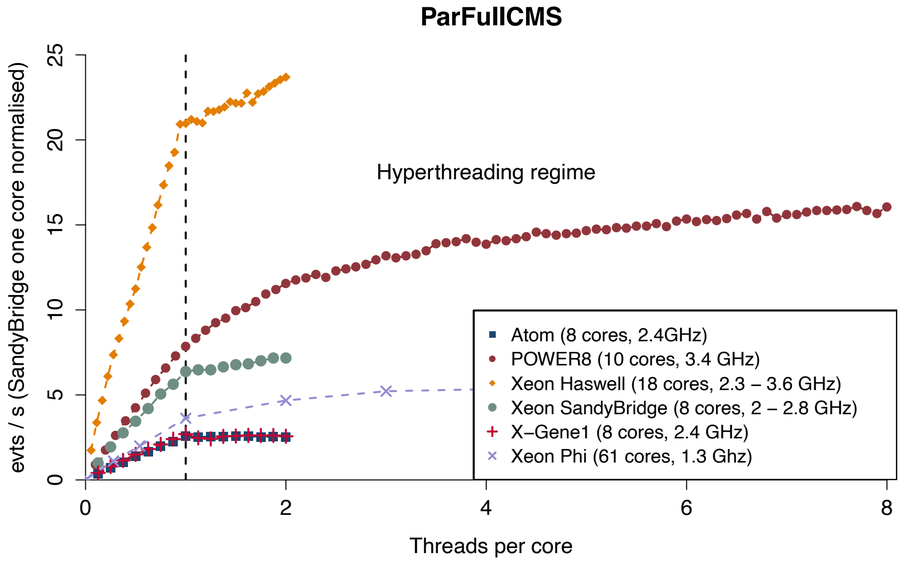
Exceptional improvements need exceptional validation!

Further benchmarking issues

- Modern CPUs can **automatically scale operating frequency** based on the workload and the environment / power consumption.
- **Turbo Boost*** from Intel (or similar for other CPUs) allows higher frequencies for a short amount of time if only a few cores are used.
- Certain features (e.g. **AVX2***vector instructions) introduce further frequency scaling.
- **Hyperthreading*** allows exploiting memory access latencies to run multiple threads concurrently on the same CPU. This resulting in boosted multithreaded performance.



David Abdurachmanov *et al* 2015 *J. Phys.: Conf. Ser.* **664** 092007



* Intel trademark, other CPU vendors offer similar features.

Understand your environment!

Modern (since the 90s) CPUs are complex beasts:

- ***Superscalar architectures, vectorisation***
- ***Aggressive frequency scaling***
- ***Simultaneous multithreading*** *(SMT, e.g. Intel Hyper Threading)*
- ***Non-Uniform Memory Access*** *(NUMA)*
- ***Non-heterogeneous architectures*** *(e.g. ARM big.LITTLE)*

Benchmarking requires **good modelling of your production environment**. *Running benchmarks on a single core might bias results compared to production running on the whole machine.*

- Elephant in the room: **GPUs**

Benchmarking tools

- What we have seen so far are common moving parts of benchmarking
- In order to help us with those common pitfalls and to allow us to write effective benchmarks, there is a number of packages which can help us. Most common are:
  - Catch2
  - Google Benchmark
- Both libraries are part of a full blown testing suite
- We will use Catch2 as it's extremely easy to setup and use. For more advanced usage, you might want to use Google Benchmark.

Catch2 example

```cpp
#include "catch_amalgamated.hpp"

#include <cstdint>

uint64_t fibonacci(uint64_t n) {
    return number < 2 ? number : fibonacci(n - 1) + fibonacci(n - 2);
}

TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
    REQUIRE(fibonacci(5) == 5);

    REQUIRE(fibonacci(20) == 6'765);
    BENCHMARK("fibonacci 20") {
        return fibonacci(20);
    };

    REQUIRE(fibonacci(25) == 75'025);
    BENCHMARK("fibonacci 25") {
        return fibonacci(25);
    };
}
```

Many testing frameworks can also provide benchmarking features.

This allows to treat correctness and performance at the same level.

However, benchmarks are usually disabled by default to limit their impact on the CI.

Catch2 example

```cpp
#include "catch_amalgamated.hpp"

#include <cstdint>

uint64_t fibonacci(uint64_t n) {
    return number < 2 ? number : fibonacci(n - 1) + fibonacci(n - 2);
}

TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
    REQUIRE(fibonacci(5) == 5);

    REQUIRE(fibonacci(20) == 6'765);
    BENCHMARK("fibonacci 20") {
        return fibonacci(20);
    };

    REQUIRE(fibonacci(25) == 75'025);
    BENCHMARK("fibonacci 25") {
        return fibonacci(25);
    };
}
```

Many testing frameworks can also provide benchmarking features.

This allows to treat correctness and performance at the same level.

However, benchmarks are usually disabled by default to limit their impact on the CI.

Catch2 example

```cpp
#include "catch_amalgamated.hpp"

#include <cstdint>

uint64_t fibonacci(uint64_t n) {
    return number < 2 ? number : fibonacci(n - 1) + fibonacci(n - 2);
}

TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
    REQUIRE(fibonacci(5) == 5);

    REQUIRE(fibonacci(20) == 6'765);
    BENCHMARK("fibonacci 20") {
        return fibonacci(20);
    };

    REQUIRE(fibonacci(25) == 75'025);
    BENCHMARK("fibonacci 25") {
        return fibonacci(25);
    };
}
```

Code within the BENCHMARK macro is executed multiple times

One single execution is usually called **run** or **iteration**

Multiple **iterations** can be grouped together in what is usually referred to as **sample**

Per benchmark execution, the **number of iterations in a sample stays constant**

The **`return`** statement prevents optimising the result

Catch2 example

```
Filters: "Benchmark Fibonacci"
Randomness seeded to: 3557866120

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fibonacci is a Catch2 v3.4.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
Benchmark Fibonacci
-------------------------------------------------------------------------------
fibonacci.cxx:9
.....................................................................

benchmark name                       samples       iterations    estimated
                                     mean          low mean      high mean
                                     std dev       low std dev   high std dev
-------------------------------------------------------------------------------
fibonacci 20                                 100         54304          0 ns
                                       0.683334 ns   0.682429 ns    0.68476 ns
                                      0.00570411 ns 0.00405876 ns 0.00785743 ns

fibonacci 25                                 100         54916          0 ns
                                       4.43742 ns    4.42027 ns    4.47205 ns
                                       0.119296 ns  0.0727883 ns   0.226172 ns

===============================================================================
All tests passed (3 assertions in 1 test case)
```

Results are provided at the end of the run.

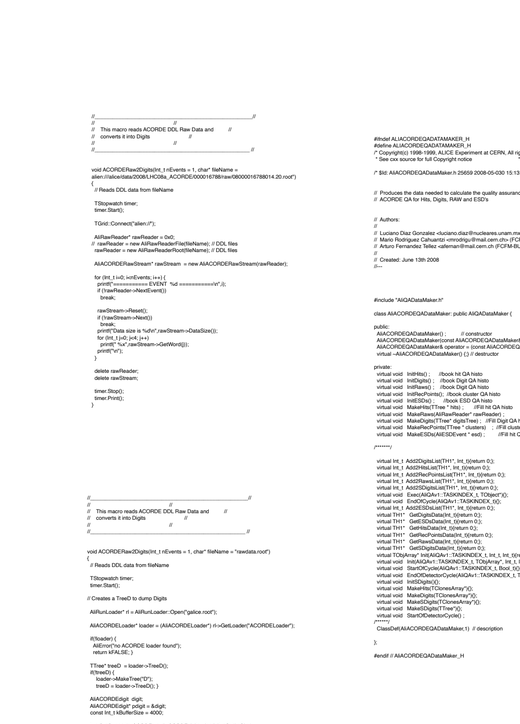
One can often provide a seed to enforce **reproducibility**.

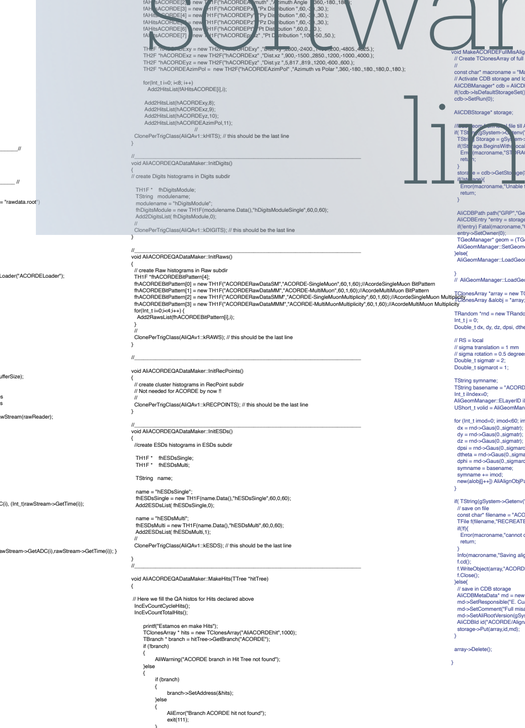
Before the actual benchmarking happens, an initial **estimation** of the result happens. Such estimation might warm up the caches.

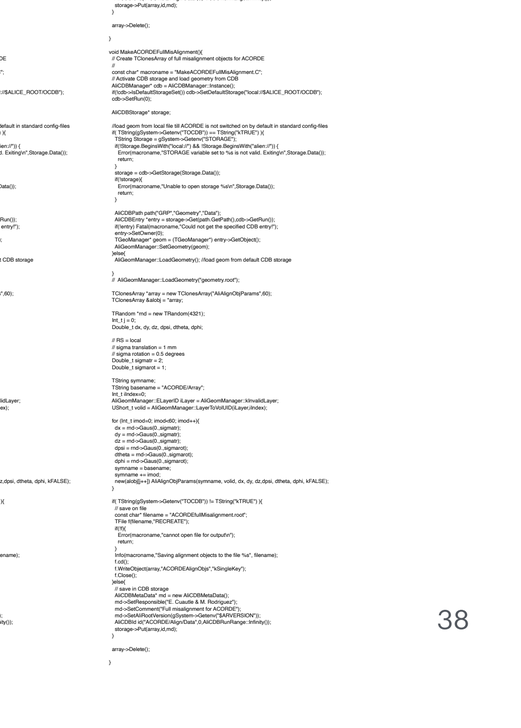
The actual **measurement** consists of a given number of samples collected one after the other.

```cpp
#include "catch_amalgamated.hpp"

#include <cstdint>

uint64_t fibonacci(uint64_t n) {
    return number < 2 ? number : fibonacci(n - 1) + fibonacci(n - 2);
}

TEST_CASE("Benchmark Fibonacci", "[!benchmark]") {
    REQUIRE(fibonacci(5) == 5);

    REQUIRE(fibonacci(20) == 6'765);
    BENCHMARK("fibonacci 20") {
        return fibonacci(20);
    };

    REQUIRE(fibonacci(25) == 75'025);
    BENCHMARK("fibonacci 25") {
        return fibonacci(25);
    };
}
```

Congratulations! You have obtained the "Fibonacci Master" Badge







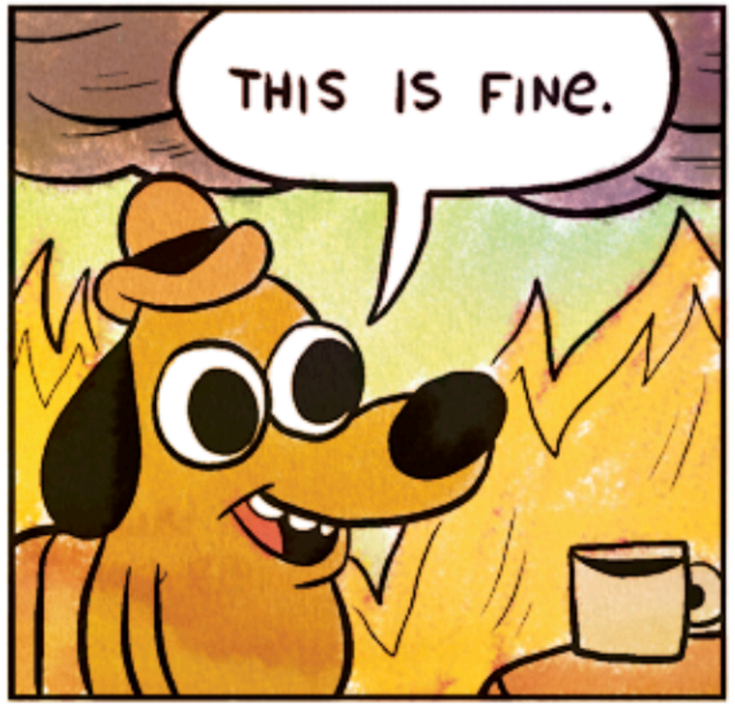
HEP Software

Writing software for an HEP experiment nowadays is a large and complex team effort:

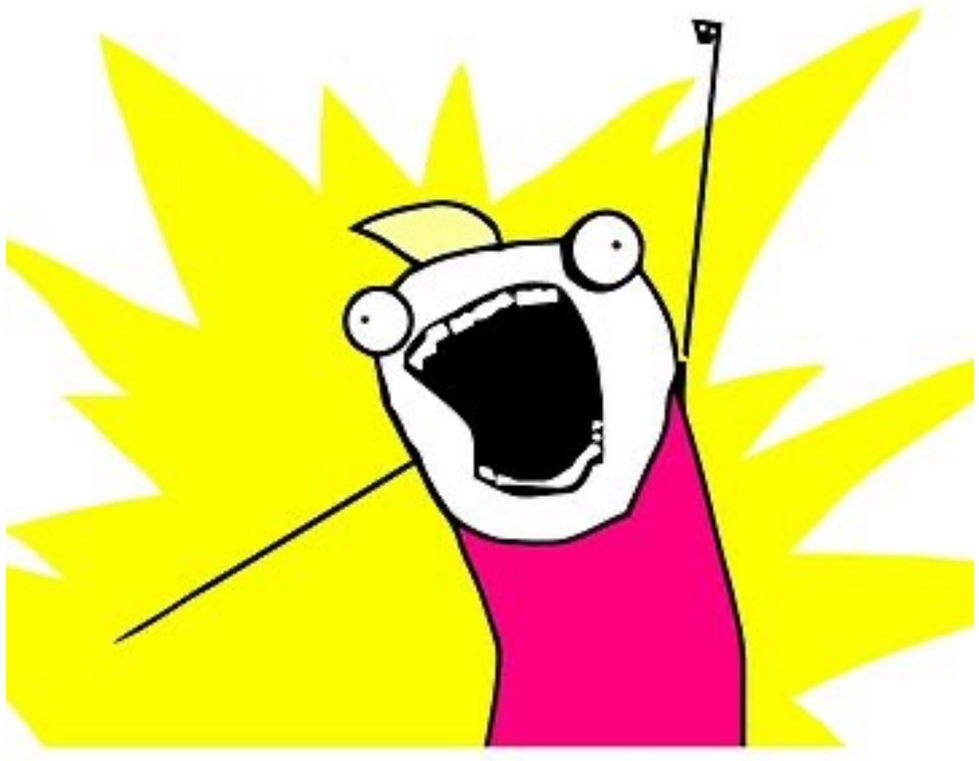
- **Large in house codebases** (O(1M) lines of code) from hundreds of diverse collaborators
- Large number of large **external dependencies** (O(10M) lines of code)
- Some very **low-level components** (readout drivers, multithreaded frameworks, patched linux kernels)
- Some **closed source** components (GPU runtime, HW drivers, generators)

Memory usage is sometimes a stricter requirement than CPU usage.

While important to model our problems, benchmarking can only bring us up to a certain point. **We need tools which are able to cope with the complexity of our environment**.



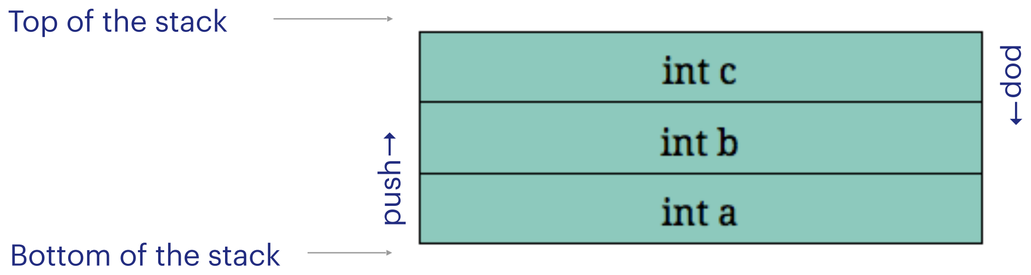
Profile all the things!



Preliminaries: the stack

Each program has an area of memory which is used to store (push) variables local to a given function. This area is known as **the stack**.

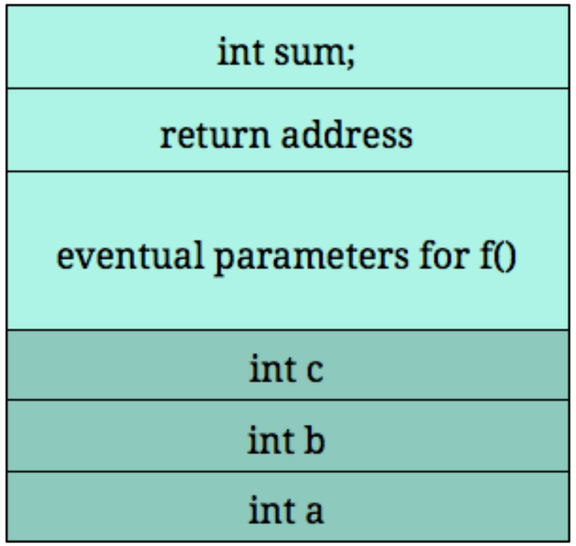
```
int main (…) {
  int a;
  int b;
  int c;
}
```



Preliminaries: the stack

When a function is called the information to handle the call can* be saved on the stack, forming a so called **stack-frame**.

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```

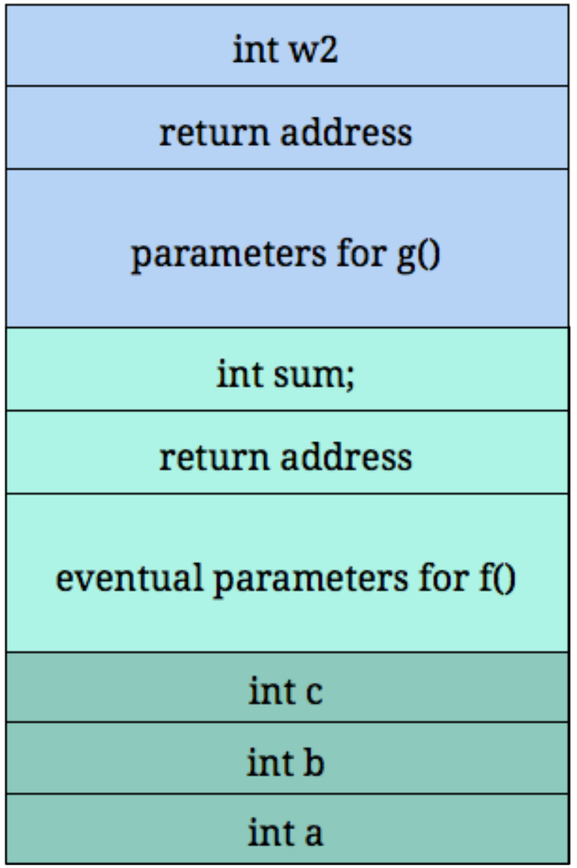


*: what actually ends up on the stack is a complicated matter as it is architecture specific and moreover the compiler can do a number of specific optimisations.

Preliminaries: the stack

Nested functions add up one on top of the other in what is called a **callstack**.

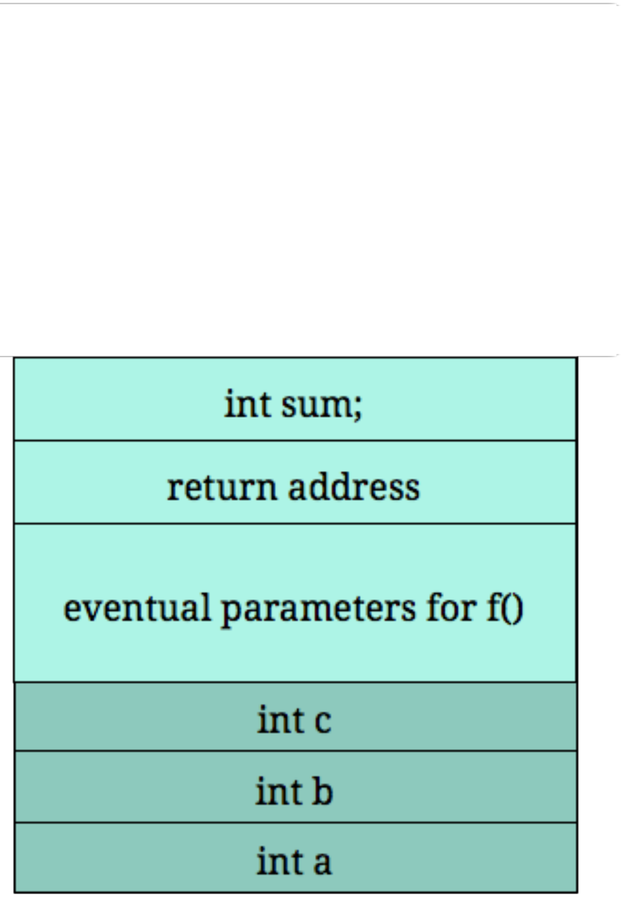
```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



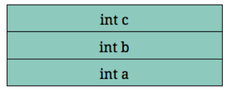
Preliminaries: the stack

Whenever a function returns, the frame gets "**pop-ed**" from the stack.

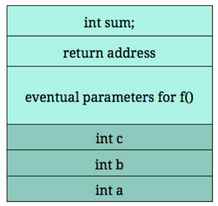
```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



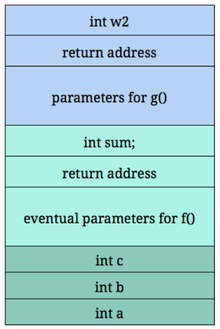
```
int main (…) {
  int a;
  int b;
  int c;
}
```



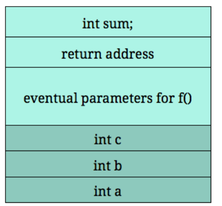

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



The whole process happens for **each function** call.

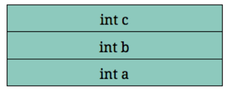
External tools (like debuggers or profilers) can be **injected** in the program flow to inspect the current callstack.

The callstack gives us a **precise location** on where we are inside our program, giving us a powerful tool to understand large codebase, with or without the sources.

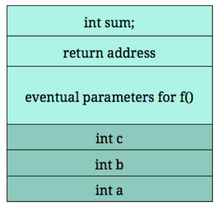
If we were able to **accumulate the callstacks** we can analyse the full program flow as it happens.

How can we do so?

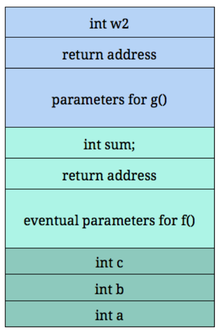
```
int main (…) {
  int a;
  int b;
  int c;
}
```



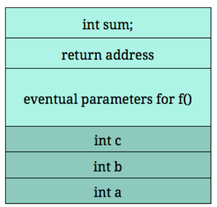

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



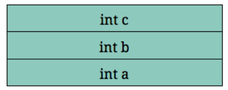

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



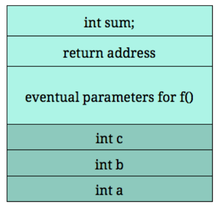

```
> lldb -- callstack-example
```

*One set of tools which allows us do so are the debuggers like **gdb** (Linux) or **lldb** (macOS)*

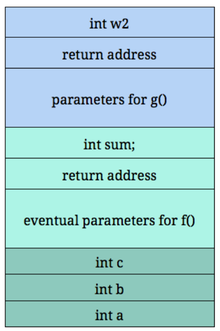
```
int main (…) {
  int a;
  int b;
  int c;
}
```



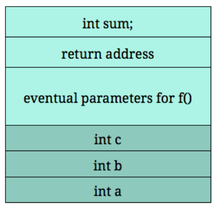

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



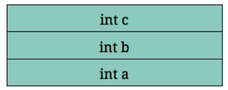

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



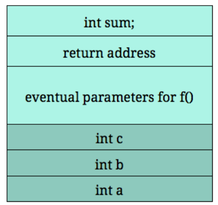

```
> lldb -- callstack-example

(lldb) b g
Breakpoint 1: where = callstack-ex`g(int) + 8 at cs.cc:2:12, address = 0x100003ee8
```

Let's set a **breakpoint** (**b**) to stop execution when entering **g()**.

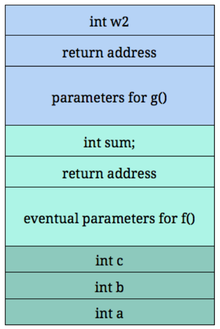
```
int main (…) {
  int a;
  int b;
  int c;
}
```



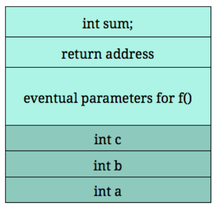

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



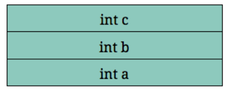

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```



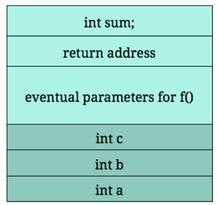

```
> lldb -- callstack-example

(lldb) b g
Breakpoint 1: where = callstack-ex`g(int) + 8 at cs.cc:2:12, address = 0x100003ee8

(lldb) r
Process 4419 launched: '/Users/ktf/src/csc-2024/callstack-ex' (arm64)
Process 4419 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x100003ee8 callstack-ex`g(w=12) at cs.cc:2:12
   1    int g(int w) {
-> 2      int w2 = w*w;
   3      return w2;
   4    }
   5
   6    int f(int x, int y,  int z) {
   7      int sum = x + y + z;
Target 0: (callstack-ex) stopped.
```

Execution will stop at the beginning of **g**. The debugger used the stackframe to provide the **frame #0** information.

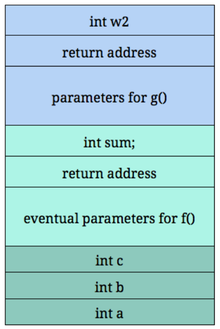
```
int main (…) {
  int a;
  int b;
  int c;
}
```



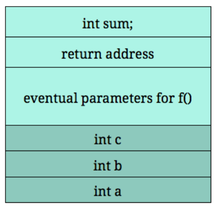

```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
g(int w) {
  int product w2 = w*w;
  return w2;
}

f(int x, int y, int z) {
  int sum = x + y + z;
  return g(sum);
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```




```
f(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}

int main (…) {
  int a;
  int b;
  int c;
  f(a, b, c);
}
```

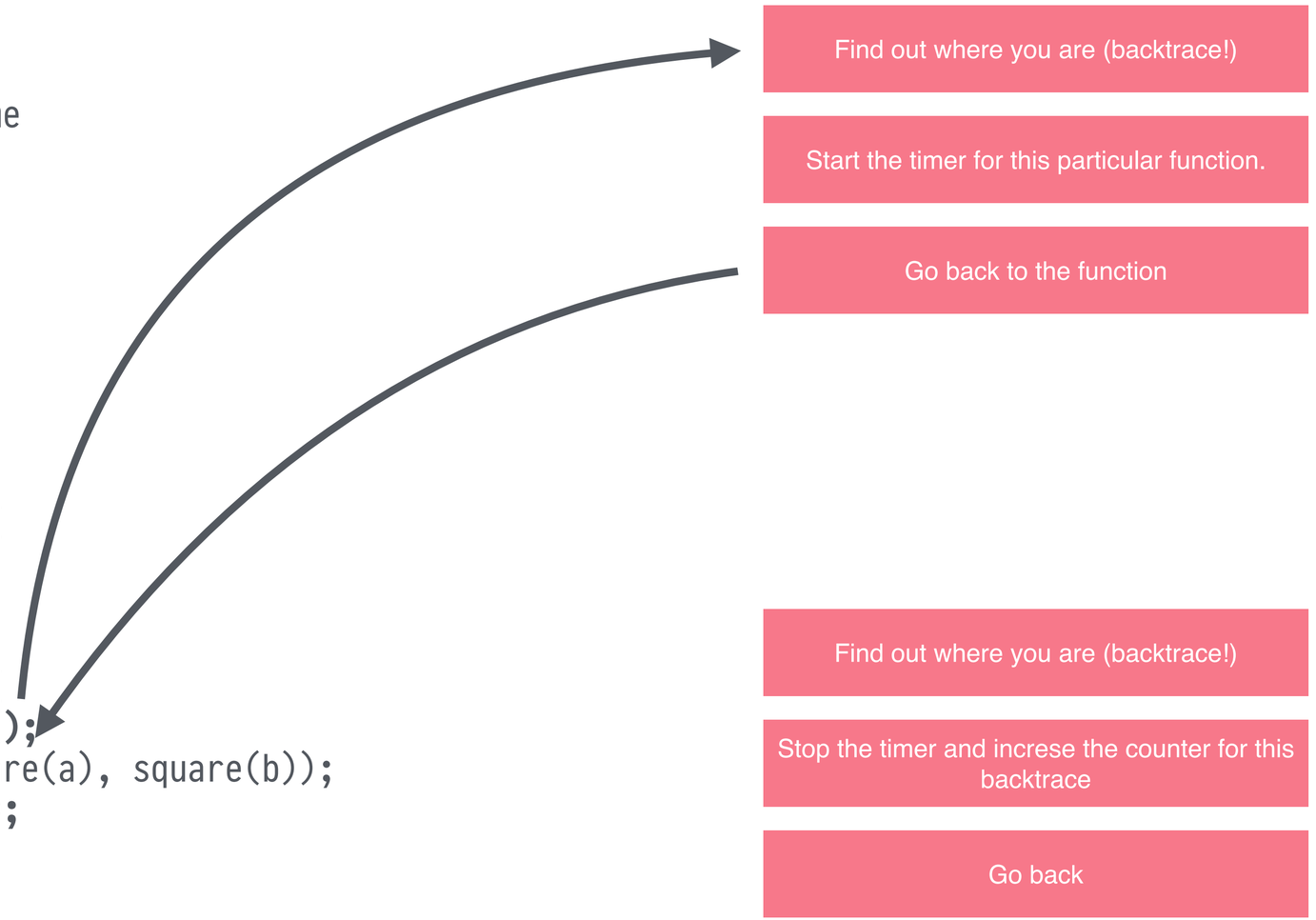



```
> lldb -- callstack-example

(lldb) b g
Breakpoint 1: where = callstack-ex`g(int) + 8 at cs.cc:2:12, address = 0x100003ee8

(lldb) r
Process 4419 launched: '/Users/ktf/src/csc-2024/callstack-ex' (arm64)
Process 4419 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    f
    1
-> 2
    3
    4
    5
    6
    7
Target 0: (callstack-ex) stopped.

(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  * frame #0: 0x0000000100003ee8 callstack-ex`g(w=12) at cs.cc:2:12
    frame #1: 0x0000000100003f3c callstack-ex`f(x=3, y=4, z=5) at cs.cc:8:10
    frame #2: 0x0000000100003f90 callstack-ex`main(argc=1, (null)) at cs.cc:15:3
    frame #3: 0x000000018cfde0e0 dyld`start + 2360
```

bt does the **backtrace** and shows the current callstack

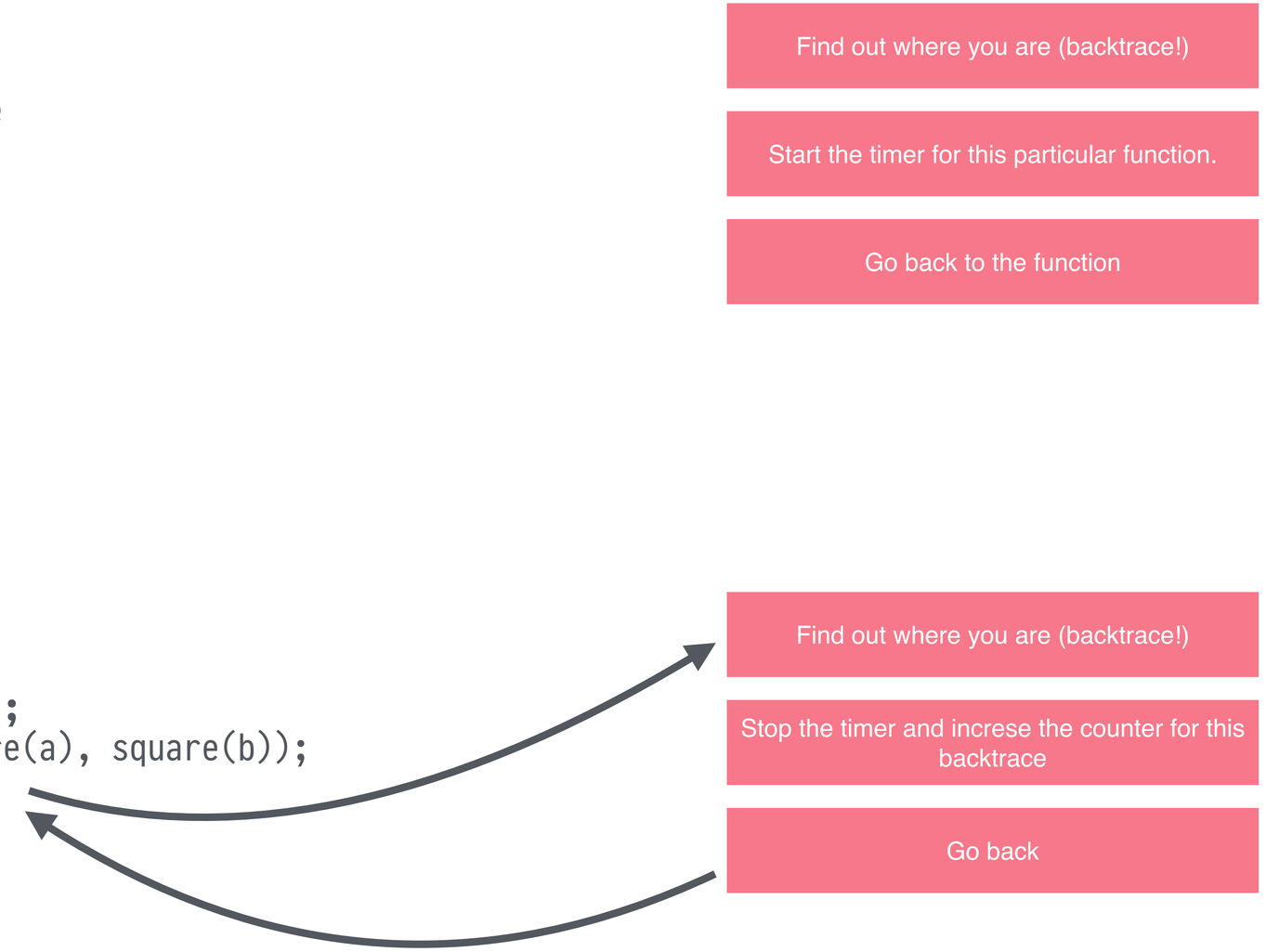
Instrumentation

```
int sqrt(int a) {
  int i = 0;
  // Don't do this @home
  while(i++ > 0)
    if (i*i >= a)
      return i;
  return 0;
}

int square (int a) {
return a*a;

int sum(int a, int b) {
  return a+b;
}

int hyp(int a, int b) {
  instrumentationStart();
  int result = sum(square(a), square(b));
  instrumentationStop();
  return result;
}
```

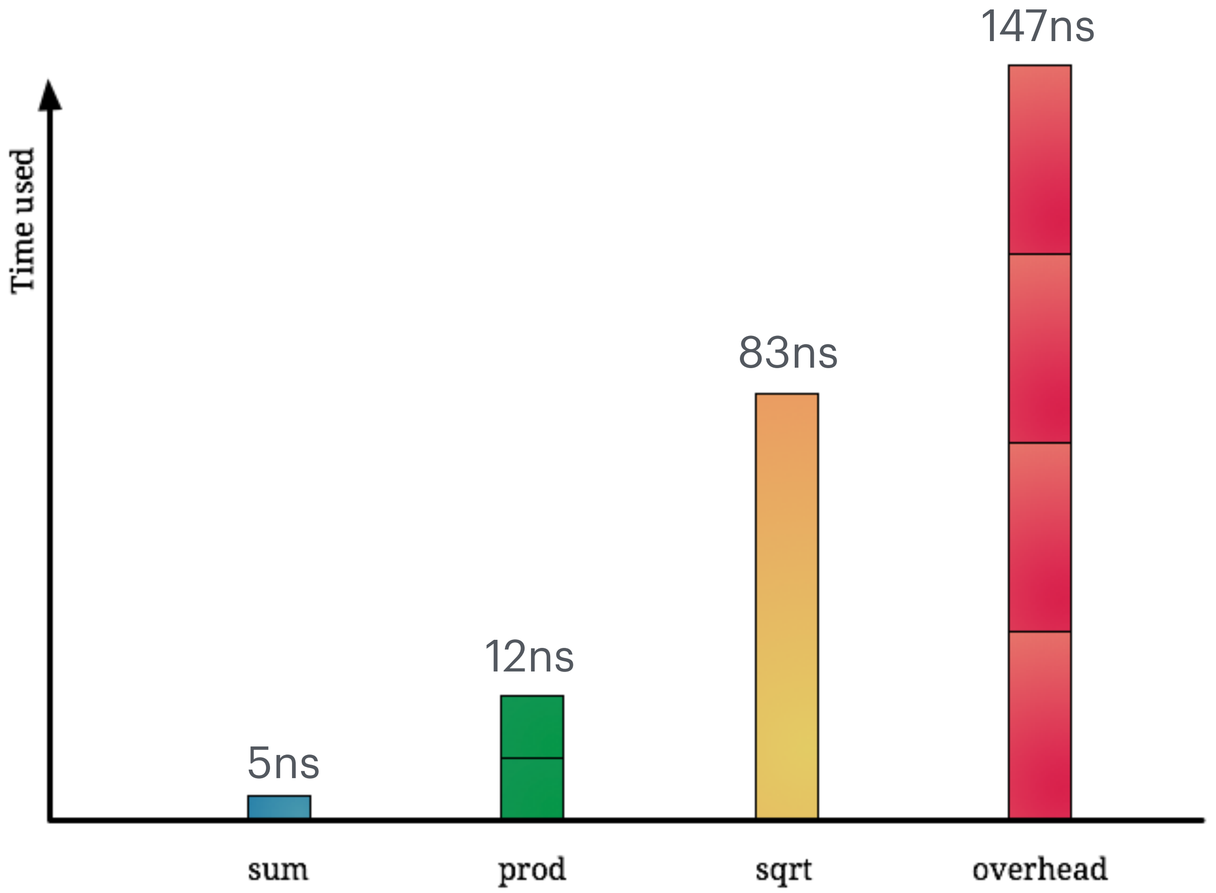


Instrumentation

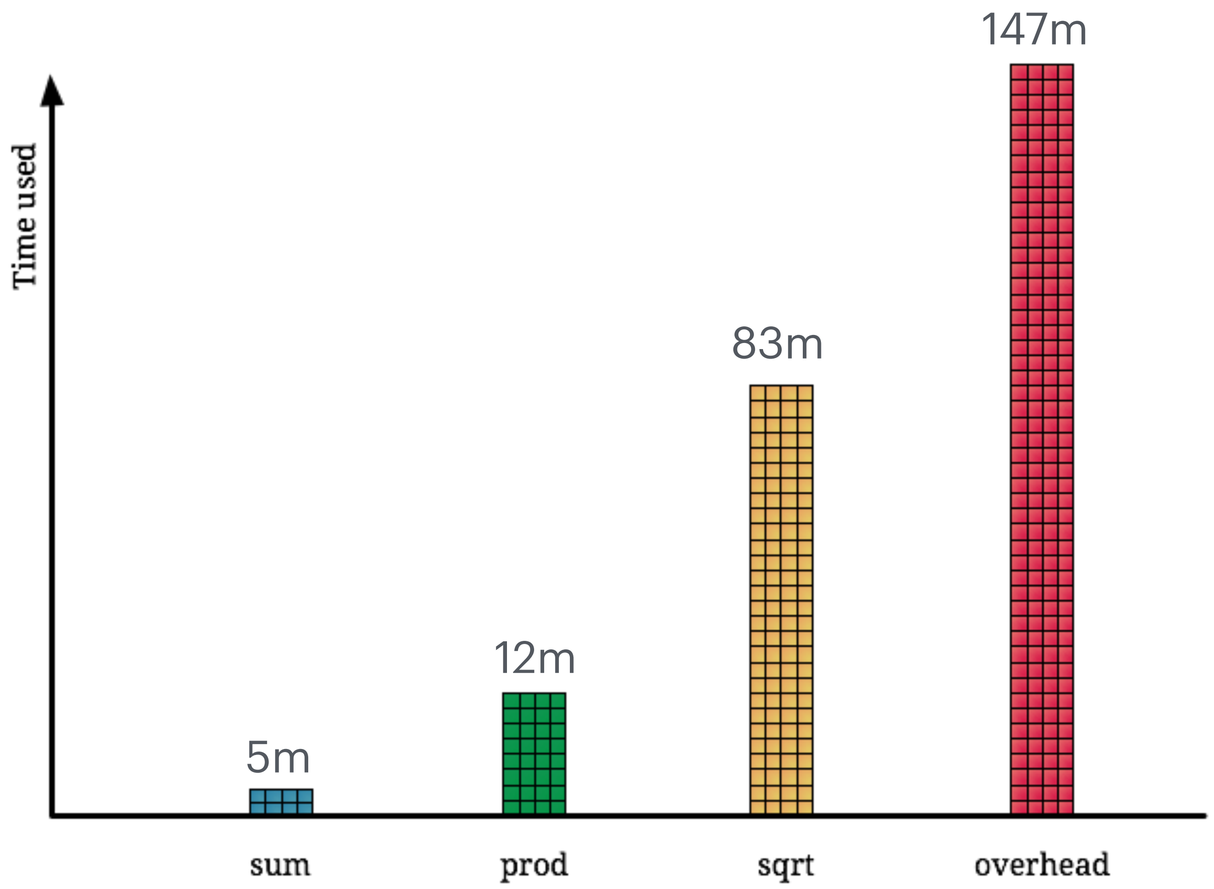
```
int sqrt(int a) {
  int i = 0;
  // Don't do this @home
  while(i++ > 0)
    if (i*i >= a)
      return i;
  return 0;
}

int square (int a) {
return a*a;

int sum(int a, int b) {
  return a+b;
}

int hyp(int a, int b) {
  instrumentationStart();
  int result = sum(square(a), square(b));
  instrumentationStop();
  return result;
}
```



Instrumentation overhead

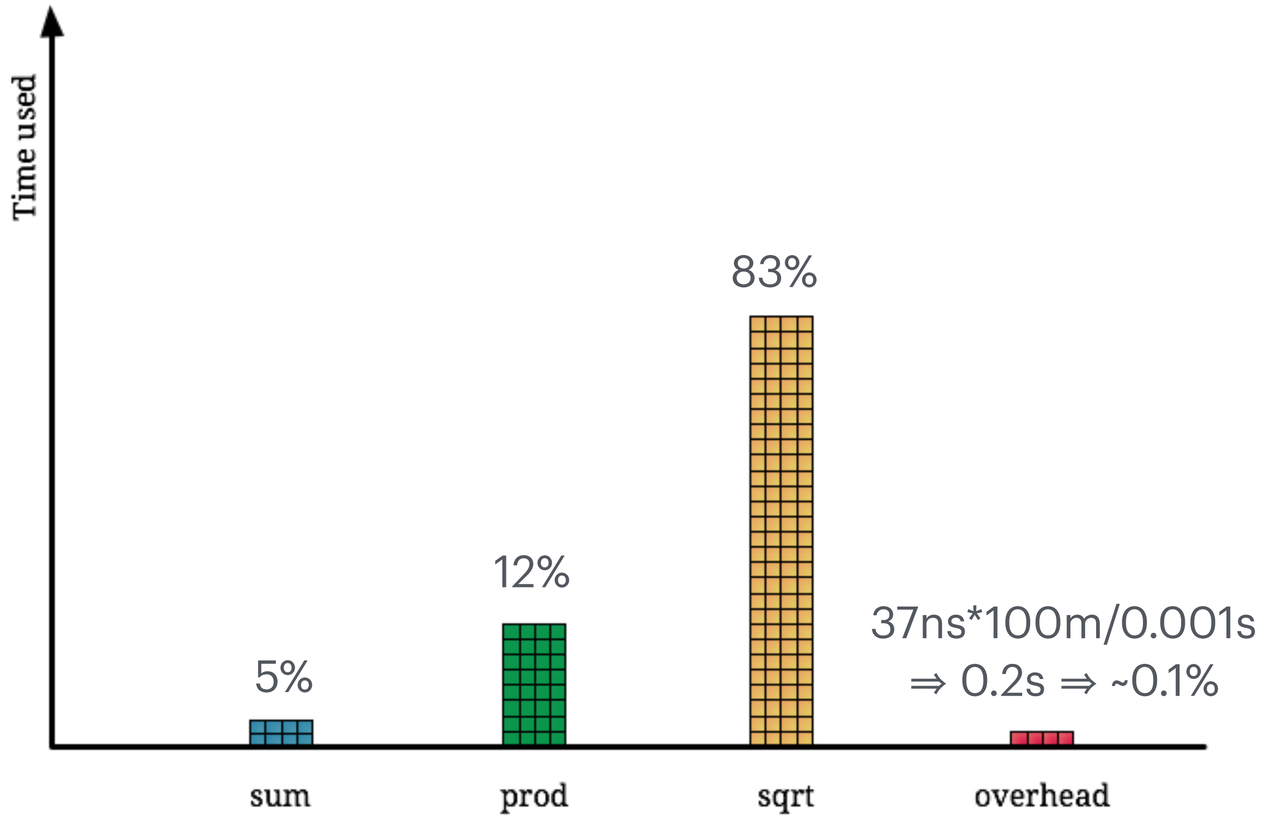


Sampling profilers



In practice, our programs have a **repetitive behaviour, lasting very long**

Sampling profilers



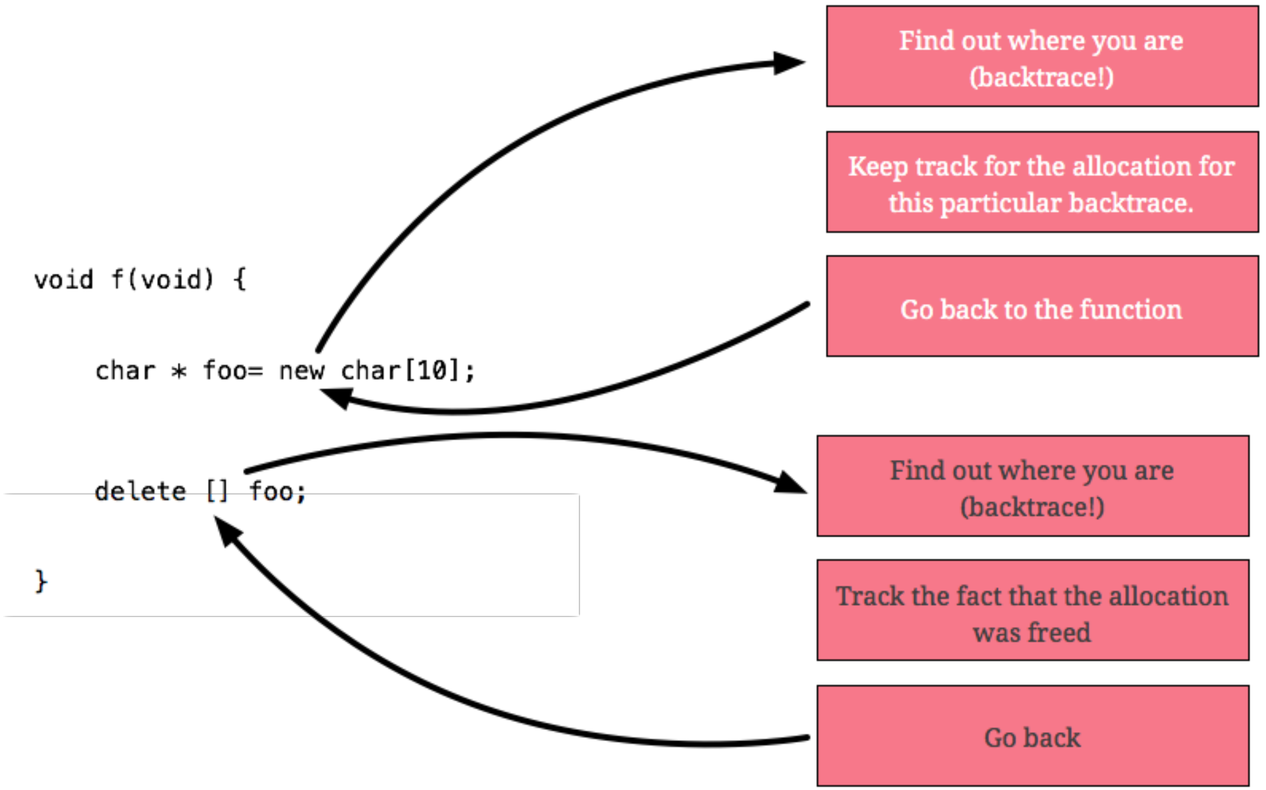
Periodically sampling every few milliseconds what is being run by the CPU **we converge** to the same time distribution while **amortising the overhead**

Memory profilers

- Sometimes the issue is not to be fast enough, but to actually fit in memory
- Memory issues can be daunting
- Memory profilers help us also to improve correctness / finding leaks
- Memory allocation is very often on the critical path for performance. Doing an allocation is expensive!

Memory profilers

Instrumentation is however very handy to collect information from smaller subsets of function. E.g. you can **track memory usage** by instrumenting malloc / free and related functions.



IgProf

Hooks into **malloc** & c

It only profiles heap allocations do not expect your 64MB array on stack to popup.

Three different kind of counters:

- **MEM_TOTAL**: sum of allocations in a call-path
- **MEM_LIVE**: sum of allocations from a given call-path, still present when profile dumps results
- **MEM_MAX:** largest **single** allocation in call-path

For each counter we store the number of calls and the allocated bytes. “Peak” mode also available.

Running IgProf

```
cat << EOF > simple-alloc.cc
#include <cstdlib>

int main(int argc, char **argv) {
  for (int n = 1; n <= 10; ++n)
    malloc(1);
}
EOF
c++ -O2 -o simple-alloc simple-alloc.cc
```

Let's consider a simple program which just allocates some memory and compile it

Running IgProf

Prefix what you want to profile with `igprof`

```
> igprof -o output.gz -mp ./simple-alloc
> igprof-analyse -g -d -r MEM_LIVE output.gz
```

Running IgProf

```
igprof -o output.gz -mp ./simple-alloc
igprof-analyse -g -d -r MEM_LIVE output.gz
```

`-o output.gz` to specify the output

Running IgProf

```
igprof -o output.gz -mp ./simple-alloc
igprof-analyse -g -d -r MEM_LIVE output.gz
```

-mp for memory profiling, **-pp** for sampling performance profiler

Run the analyser job on the profiler output to produce a report

```
> igprof -o output.gz -mp ./simple-alloc
> igprof-analyse -g -d -r MEM_LIVE output.gz
```

Analysing results

igprof-analyse

Dumping a 50MB gzipped file full of profile data is useless if you cannot extract information from it. igprof-analyse takes a profile dump and produces (somewhat) human readable reports from it. Tries hard to be accurate when doing symbol name demangling, using `nm` and `gdb`.

It also aggregates call-paths, allows applying various filters on the call-tree, both by changing contents of single nodes and by merging nodes together.

Exports results in a gprof like text file, sqlite db, or JSON object

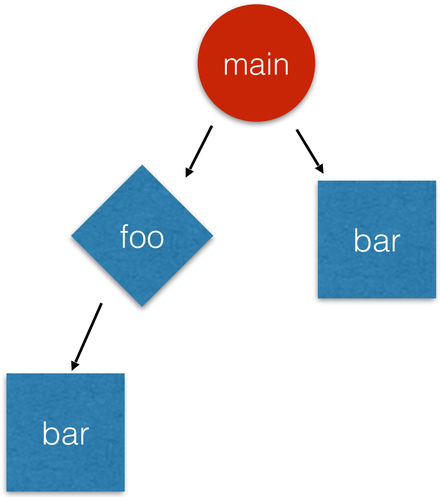
igprof-navigator

A poor man cgi script / python server which allows you to navigate results via a web interface. Uses the sqlite report as input.

Analysing results

gprof like report format

For each node in the callpath, we aggregate edge information for nodes going into the selected node (i.e. callers) and nodes going out (i.e. callees). Considered node symbol is indented to highlight it.

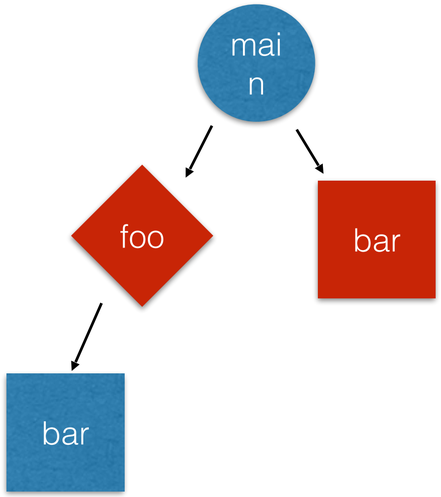



```
Rank % total      Self   Self / Children  Calls / Total   Function

[1]    100.0         4      0 / 4             3             main
        50.0  ........      2 / 3             1 / 2           bar(int) [2]
        50.0  ........      2 / 2             2 / 2           foo() [3]
```

Analysing results

gprof like report format

Lines below the selected symbol are callees.

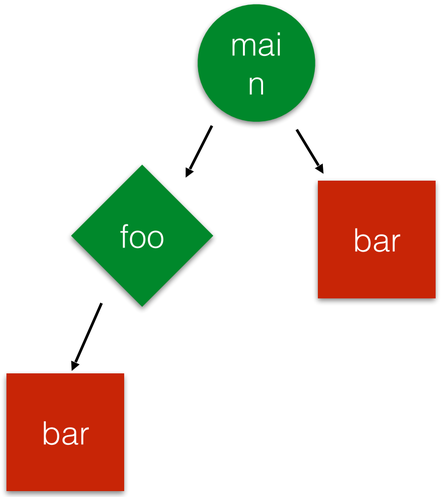



```
Rank % total      Self   Self / Children  Calls / Total   Function

[1]    100.0         4      0 / 4             3             main
        50.0  ........      2 / 3             1 / 2           bar(int) [2]
        50.0  ........      2 / 2             2 / 2           foo() [3]
```

Analysing results

gprof like report format

Lines below the selected symbol are callers.

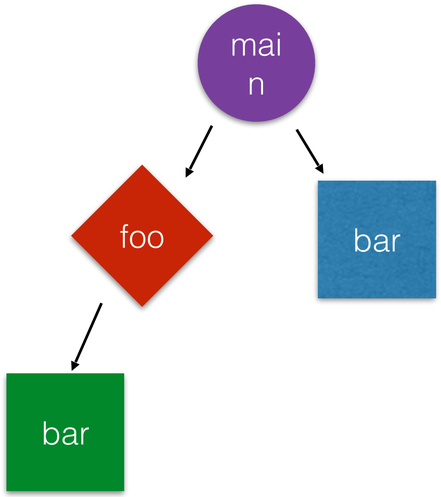



```
Rank % total      Self   Self / Children  Calls / Total   Function

        25.0  ........      1 / 2             1 / 2           foo() [3]
        50.0  ........      2 / 4             1 / 3           main [1]
[2]     75.0         3      3 / 0             2             bar(int)
```

Analyzing results

gprof like report format

Lines below the selected symbol are callers.




```
Rank % total      Self   Self / Children  Calls / Total   Function

        50.0  ........      2 / 4             2 / 3         main [1]
[3]     50.0         2      1 / 1             2           foo()
        25.0  ........      1 / 3             1 / 2         bar(int) [2]
```

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(1);
...
<igprof dumps report>
```

`MEM_LIVE` and `MEM_TOTAL` are the same if the is no deallocation

	Counts	Calls	Peak
MEM_LIVE	10	10	10
MEM_TOTAL	10	10	10
MEM_MAX	1	10	1

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(1);
...
<igprof dumps report>
```

MEM_MAX tracks single allocations

	Counts	Calls	Peak
MEM_LIVE	10	10	10
MEM_TOTAL	10	10	10
MEM_MAX	**1**	10	**1**

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
 malloc(n);
...
<igprof dumps report>
```

Counts are the sum of allocations, calls are how many times we called `malloc`

	Counts	Calls	Peak
MEM_LIVE	55	10	55
MEM_TOTAL	55	10	55
MEM_MAX	10	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
  free(malloc(n));
...
<igprof dumps report>
```

Once you start deallocating the difference between `MEM_LIVE` and `MEM_TOTAL` is obvious

	Counts	Calls	Peak
MEM_LIVE	**0**	0	**10**
MEM_TOTAL	55	10	55
MEM_MAX	**10**	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
  free(malloc(n));
...
<igprof dumps report>
```

`MEM_LIVE` in peak mode gives the largest block of memory at one point in time

	Counts	Calls	Peak
MEM_LIVE	**0**	0	**10**
MEM_TOTAL	55	10	55
MEM_MAX	**10**	10	10

Memory profiling

```
<igprof started here>
...
for (int n = 1; n <= 10; ++n)
{
  void *x = malloc(1);
  <igprof dumps report*>
  free(x);
}
...
```

Dumps can be triggered by calling a public symbol or by writing to a file specified by the `-D` option. In this particular case you get n reports.

`MEM_LIVE` becomes a "*likely leaks*" checker!

	Counts	Calls	Peak
MEM_LIVE	n	n	n
MEM_TOTAL	Σ1..n	Σ1..n	Σ1..n
MEM_MAX	n	n	n

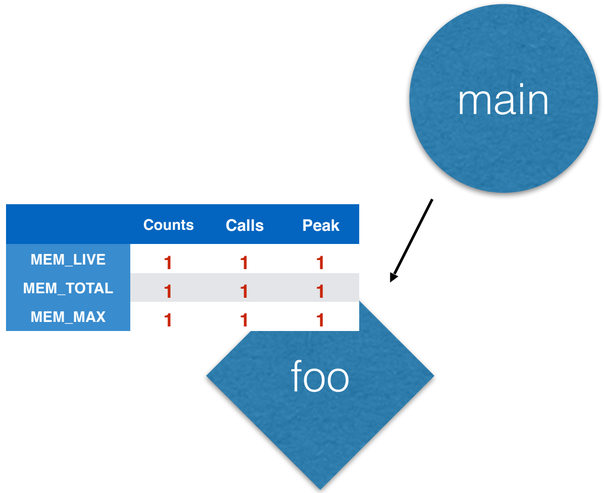
Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```

In large programs what is actually interesting is to know not only which functions allocated most of the memory, but why.

Call tree

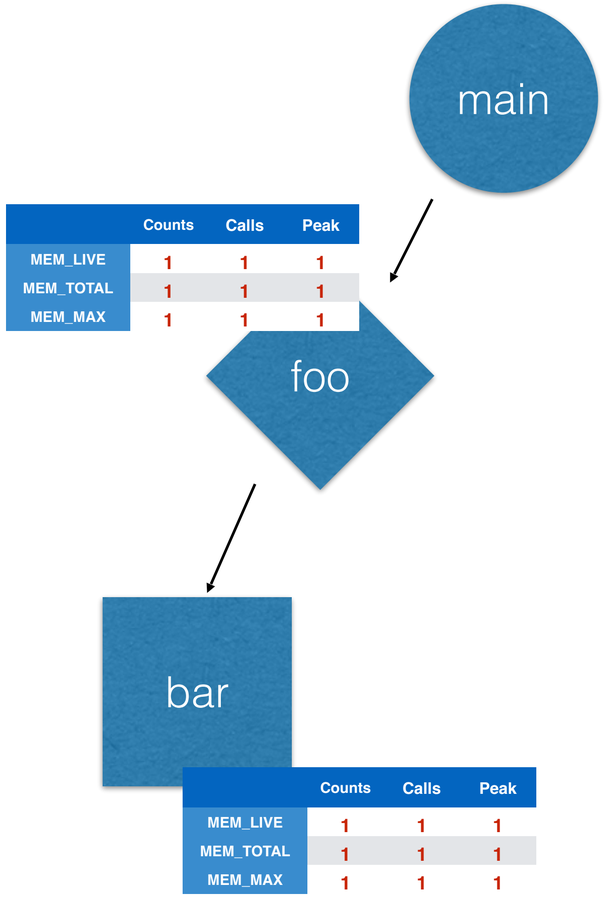
```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



Every time an instrumented function is called we determine the full calltree.

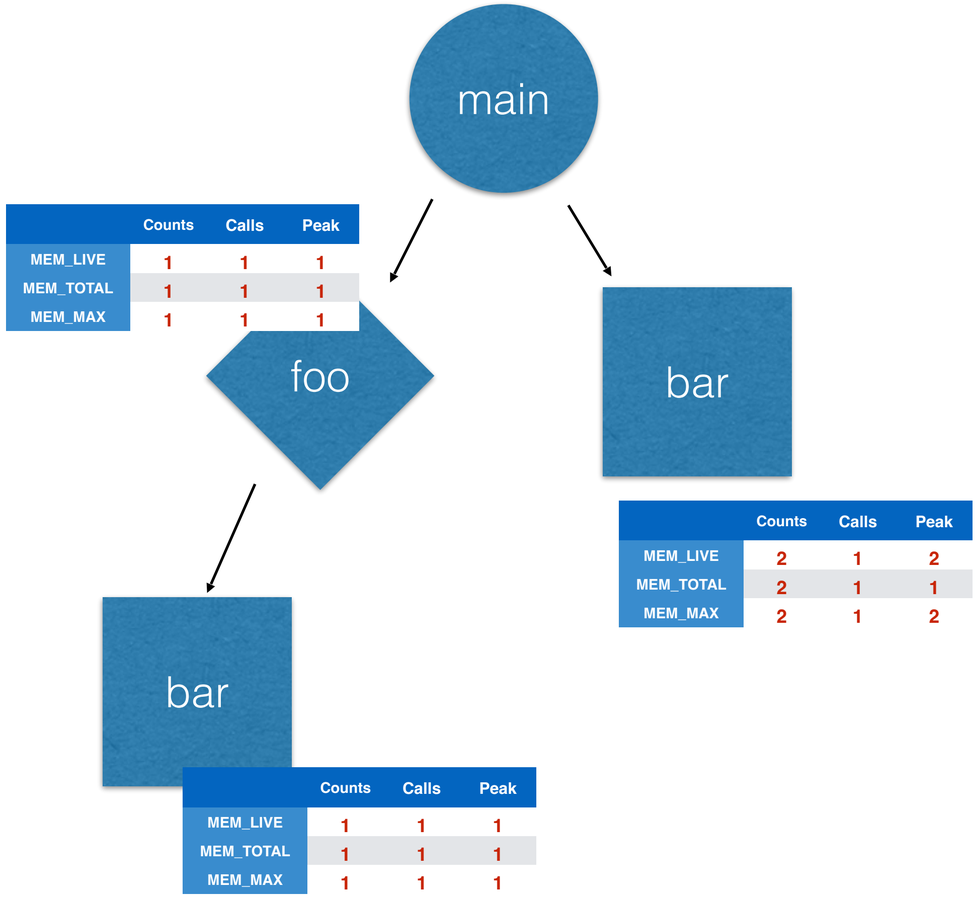
Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



Call tree

```
void bar(int i)
{
 malloc(i);
}

void foo()
{
 malloc(1);
 bar(1);
}

int main(int, char **)
{
 foo();
 bar(2);
}
```



Visualising results

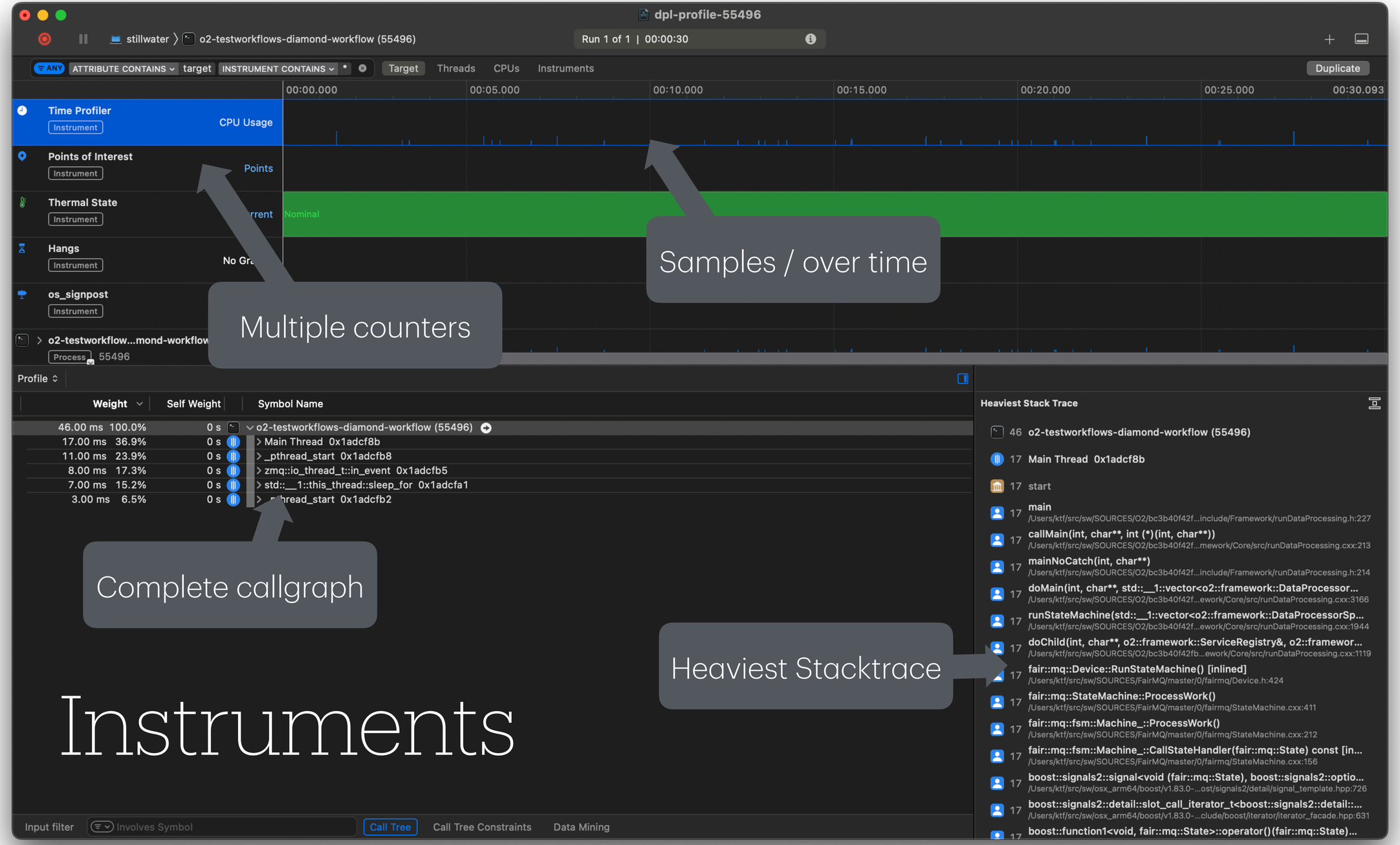
IgProf itself has a simply embedded web GUI called **IgProf Navigator** (works with python cProfile as well).

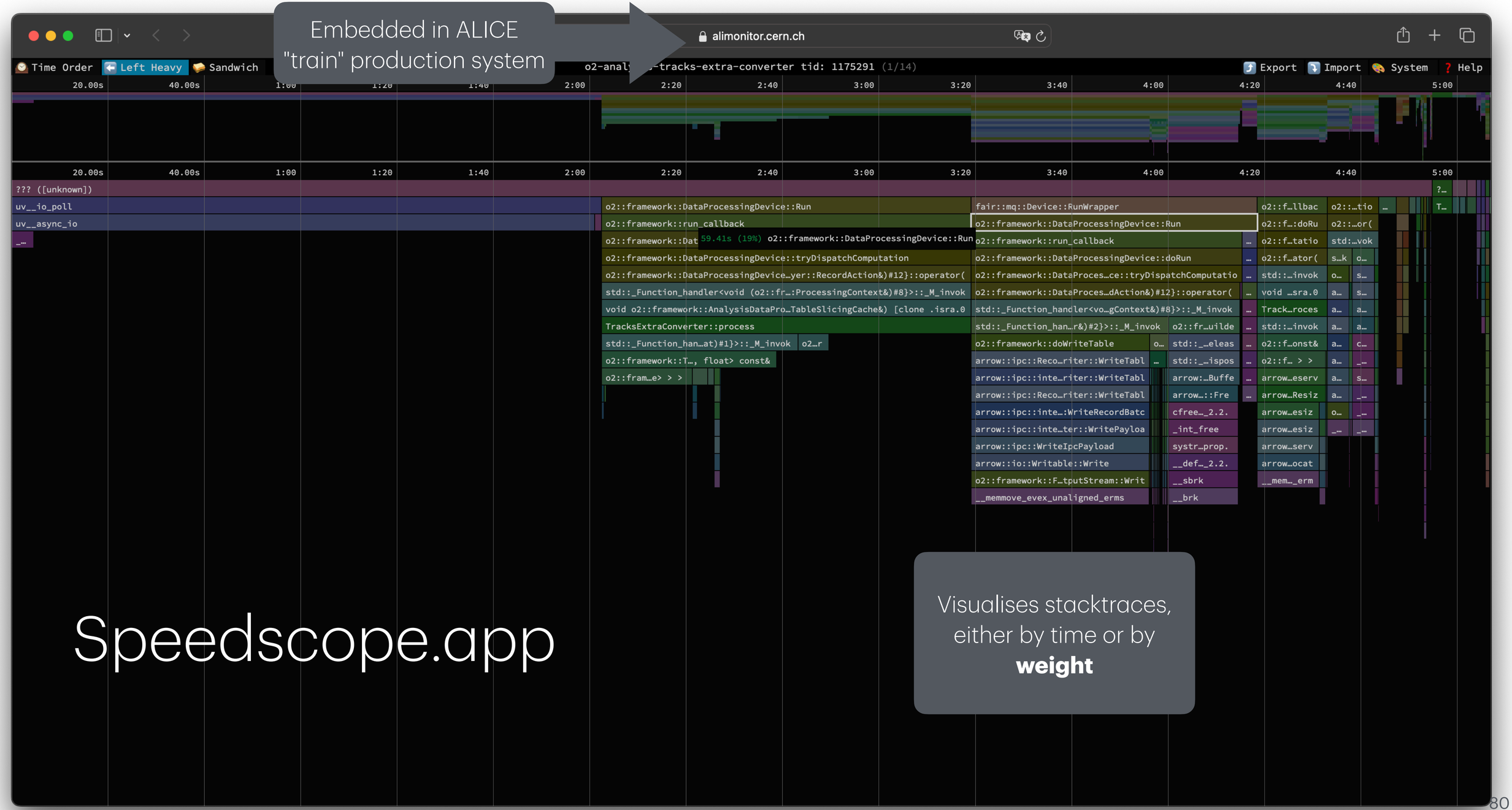
Many tools (most notably, perf) can export information in the so called **Google Trace Event** format. Results can then be visualised graphically in Chrome **about:tracing** or in a number of other tools:

- https://www.speedscope.app/
- https://profiler.firefox.com/

A number of IDEs have their own GUI:

- **Instruments**
- **VTune**


dpl-profile-55496
stillwater
o2-testworkflows-diamond-workflow (55496)
Run 1 of 1 | 00:00:30
Target
Threads
CPUs
Instruments
Duplicate
00:00.000
00:05.000
00:10.000
00:15.000
00:20.000
00:25.000
00:30.093
Time Profiler
Instrument
CPU Usage
Points of Interest
Instrument
Points
Thermal State
Instrument
Nominal
Hangs
Instrument
os_signpost
Instrument
Process
55496
Multiple counters
Samples / over time
Profile
Weight
Self Weight
Symbol Name
46.00 ms 100.0% 0 s o2-testworkflows-diamond-workflow (55496)
17.00 ms 36.9% 0 s Main Thread 0x1adcf8b
11.00 ms 23.9% 0 s _pthread_start 0x1adcfb8
8.00 ms 17.3% 0 s zmq::io_thread_t::in_event 0x1adcfb5
7.00 ms 15.2% 0 s std::__1::this_thread::sleep_for 0x1adcfa1
3.00 ms 6.5% 0 s
Complete callgraph
Heaviest Stacktrace
Heaviest Stack Trace
46 o2-testworkflows-diamond-workflow (55496)
17 Main Thread 0x1adcf8b
17 start
17 main
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42f...include/Framework/runDataProcessing.h:227
17 callMain(int, char**, int (*)(int, char**))
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42f...mework/Core/src/runDataProcessing.cxx:213
17 mainNoCatch(int, char**)
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42f...include/Framework/runDataProcessing.h:214
17 doMain(int, char**, std::__1::vector<o2::framework::DataProcessor...
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42f...ework/Core/src/runDataProcessing.cxx:3166
17 runStateMachine(std::__1::vector<o2::framework::DataProcessorSp...
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42f...ework/Core/src/runDataProcessing.cxx:1944
17 doChild(int, char**, o2::framework::ServiceRegistry&, o2::framewor...
/Users/ktf/src/sw/SOURCES/O2/bc3b40f42fb...ework/Core/src/runDataProcessing.cxx:1119
17 fair::mq::Device::RunStateMachine() [inlined]
/Users/ktf/src/sw/SOURCES/FairMQ/master/0/fairmq/Device.h:424
17 fair::mq::StateMachine::ProcessWork()
/Users/ktf/src/sw/SOURCES/FairMQ/master/0/fairmq/StateMachine.cxx:411
17 fair::mq::fsm::Machine_::ProcessWork()
/Users/ktf/src/sw/SOURCES/FairMQ/master/0/fairmq/StateMachine.cxx:212
17 fair::mq::fsm::Machine_::CallStateHandler(fair::mq::State) const [in...
/Users/ktf/src/sw/SOURCES/FairMQ/master/0/fairmq/StateMachine.cxx:156
17 boost::signals2::signal<void (fair::mq::State), boost::signals2::optio...
/Users/ktf/src/sw/osx_arm64/boost/v1.83.0-...ost/signals2/detail/signal_template.hpp:726
17 boost::signals2::detail::slot_call_iterator_t<boost::signals2::detail::...
/Users/ktf/src/sw/osx_arm64/boost/v1.83.0-...clude/boost/iterator/iterator_facade.hpp:631
17 boost::function1<void, fair::mq::State>::operator()(fair::mq::State)...
Input filter
Involves Symbol
Call Tree
Call Tree Constraints
Data Mining
Instruments


Embedded in ALICE
"train" production system
alimonitor.cern.ch
Time Order
Left Heavy
Sandwich
Export
Import
System
Help
20.00s
40.00s
1:00
1:20
1:40
2:00
2:20
2:40
3:00
3:20
3:40
4:00
4:20
4:40
5:00
??? ([unknown])
uv__io_poll
uv__async_io
o2::framework::DataProcessingDevice::Run
o2::framework::run_callback
o2::framework::DataProcessingDevice::tryDispatchComputation
TracksExtraConverter::process
fair::mq::Device::RunWrapper
o2::framework::DataProcessingDevice::doRun
o2::framework::doWriteTable
59.41s (19%) o2::framework::DataProcessingDevice::Run
Speedscope.app
Visualises stacktraces, either by time or by **weight**

More profiling tools

- IgProf: yours (and Lassi Tuura) truly. Both memory and performance profiling
- Google Perf Tools: similar to IgProf, can do both memory and sampling profiler
- Cachegrind: Valgrind fame profiler
- perf: low level profiler (uses CPU performance counters)
- VTune: Intel profiling suite
- Tracy: game developer oriented profiler. **Embeddable.**
- Instruments: Apple profiler, part of XCode
- cProfile: **python** embedded profiler

Inspiring final last words

Inspiring final last words

Procedurally generated in **177 kilobytes**