

PERFORMANCE-ORIENTED COMPUTING

Performance Evaluation & Tools



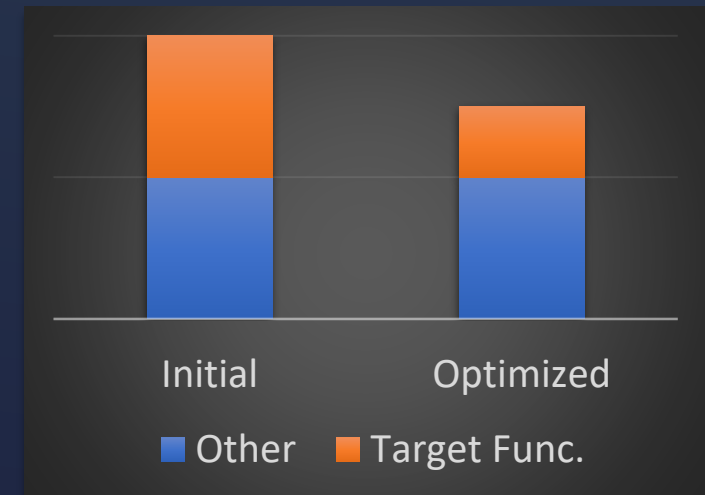
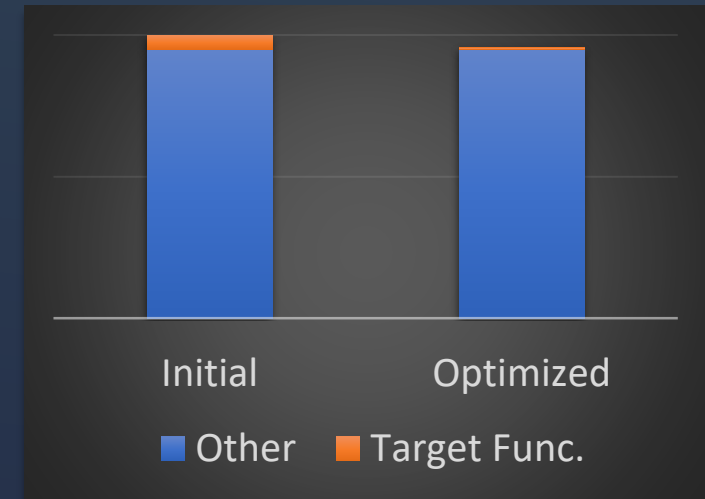
GOALS

- ▶ Improve ability to **interpret** performance results
 - ▶ General considerations when **targeting** optimizations
 - ▶ Bottleneck analysis, roofline model
- ▶ Become aware of the fundamental importance of **loops** (and recursion)
- ▶ Understand the available **tools** for performance evaluation
 - ▶ Tracing- and Sampling-based profiling
- ▶ Leverage interpretation aids **beyond time measurement**

INTERPRETATION

GENERAL IDEA

- ▶ Independently of the specific analysis and interpretation, our primary goal with performance analysis is to determine **where optimization effort can be spent most effectively**
- ▶ An example to illustrate the essential principle:
 - ▶ Function which requires **5%** of the time, optimized **10x** → program **4.7% faster**
 - ▶ Function which requires **50%** of the time, optimized **2x** → program **33% faster**



BASIC INTERPRETATION

- ▶ Perhaps the most basic analysis (but still frequently useful):
Before / after comparison
 - ▶ Allows us to monitor performance evolution during optimization
→ fundamentally important use case!
- ▶ However, provides **no** insight into the quality of an implementation in **absolute** terms
 - ▶ This requires more information concerning both the **algorithms** and **hardware** used

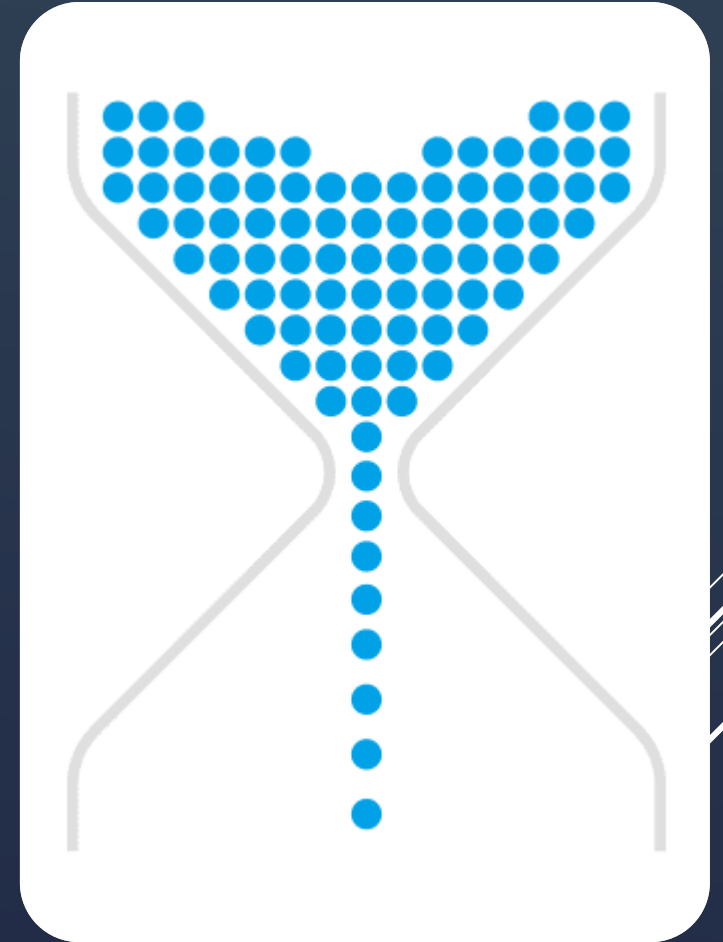
BOTTLENECK ANALYSIS

- ▶ **Goal:**

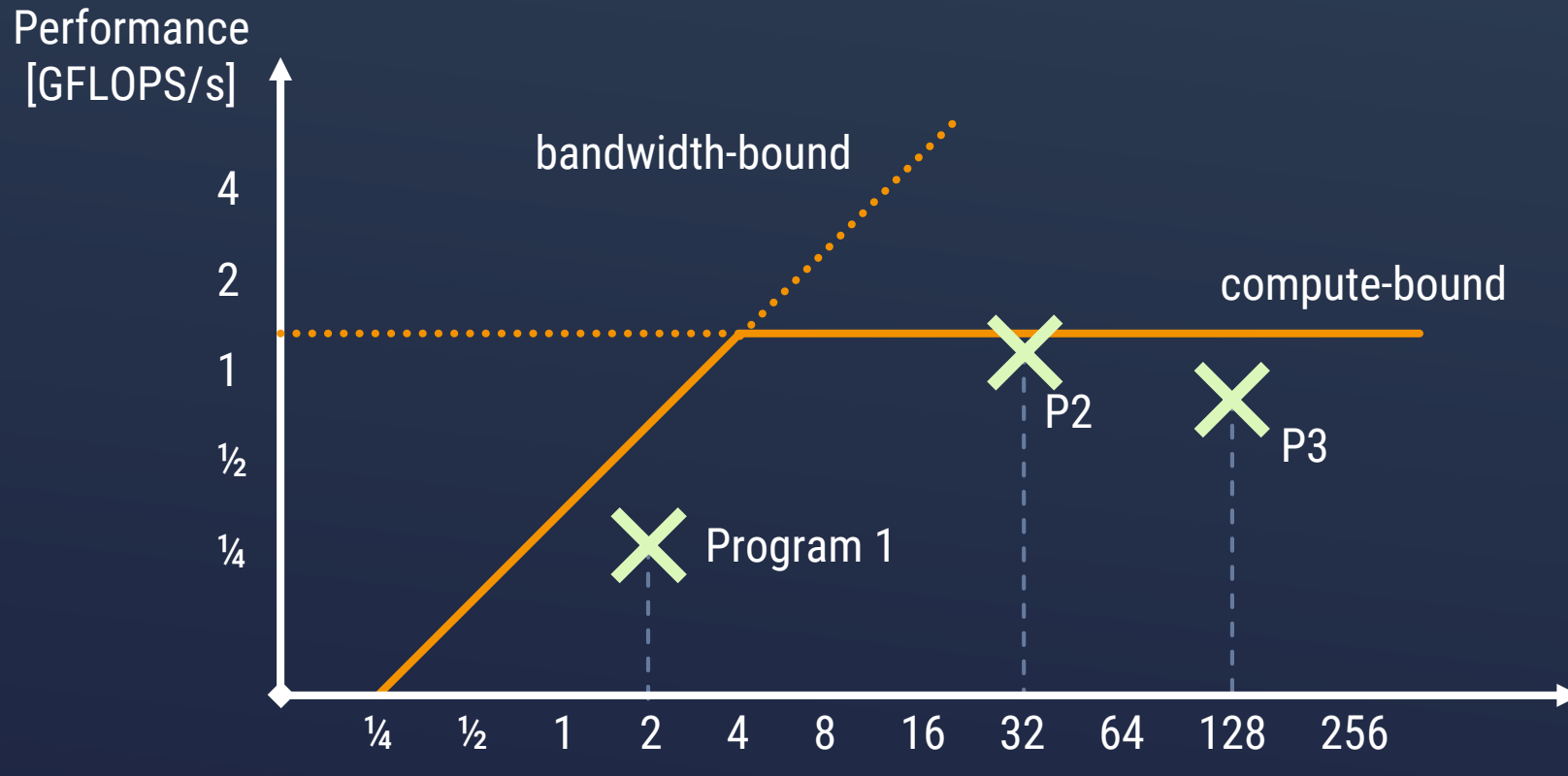
Determine which factor(s) are limiting a given program on a given platform (i.e. memory access, computation, I/O, ...)

- ***Bottleneck***

- ▶ Generally not binary – a real world application is very unlikely to be 100% memory limited, or compute limited, etc.
- ▶ Interaction of two bottleneck factors is frequently visualized using a **Roofline Model**



ROOFLINE MODEL



Various extensions used in practise:

- Parallelism
- Caching / memory hierarchy
- NUMA
- ...

operational
intensity
[FLOPS/byte]

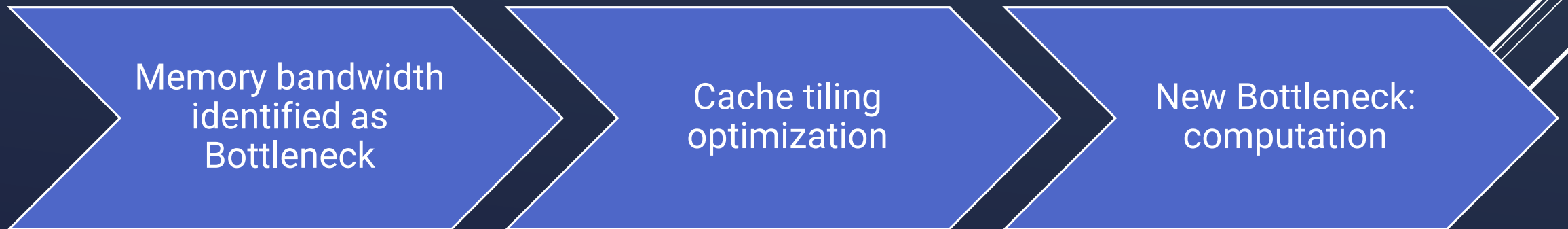
ROOFLINE MODEL – REASONING

- ▶ (Mostly) **bandwidth** limited:
 - ▶ Check / optimize cache behavior
 - ▶ If applicable: NUMA binding, software prefetching, ...
- ▶ (Mostly) **computation** limited:
 - ▶ Vectorization (SIMD)
- ▶ **Neither:**
 - ▶ Check input/output behavior
 - ▶ Access patterns or data structures Latency dependent?



BOTTLENECK IS NOT STATIC

- ▶ Bottleneck can ***change*** in multiple ways
 - ▶ During the execution of different **program phases**
 - ▶ During the **optimization process**
- ▶ Example:



THE IMPORTANCE OF LOOPS

... and recursion, but that's usually less common outside functional languages



10

BASIC RUNTIME ESTIMATES

- ▶ Assume (for simplicity):
 - ▶ A program binary size of **10 MB**
 - ▶ Fixed-sized binary instructions with a size of **8 bytes**
 - ▶ A **5 GHz** processor with an **IPC rate of 1.0** for all instructions
 - ▶ A **cache miss rate of 2%**, with each miss causing a **200 cycle** stall
 - ▶ No other sources of stalls, and no function calls
- ▶ *How long can this program take to execute, at most, if it has no loops?*

BASIC RUNTIME ESTIMATES

- ▶ $10 \text{ MB} / 8 \text{ Byte} = 1250000 \text{ Instructions} = 1250000 \text{ cycles of execution}$
- ▶ $1250000 * 0.02 * 200 = 5000000 \text{ stalled cycles}$
- ▶ $(1250000 + 5000000) / (10^9 * 5) = 0.00125 \text{ seconds}$

→ The program can run for at most **1.25 ms** if it does not contain any loops

- ▶ This would be assuming that every single instruction is executed, and a relatively high cache miss rate
- ▶ Allowing function calls (but no recursion) changes the picture – by building a binary tree we can get $O(2^{\frac{N}{2}})$ instructions in a constructed case

THE TOY EXAMPLE CASE

- ▶ In small applications and toy codes, the relevant loops can be very **obvious**:
 - ▶ At the source code level, they are spatially close to the loop body
 - ▶ At the execution level, there's likely not too much happening in one individual execution of the loop nest

```
// conduct multiplication
for (int i=0; i<N; i++) {
    for (int j=0; j<K; j++) {
        TYPE sum = 0;
        for (int k=0; k<M; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

REAL-WORLD CHALLENGES

In large-scale real-world programs, the relevant loops will frequently be:

1. **Harder to identify**: they might contain a lot more code, and the total execution time might not be dominated by any single loop nest
2. **Spread out** at the code level: an important loop might easily call different functionality from several distinct modules spanning many source files
3. Part of some **deep nesting**, with several dozen levels of function calls above or below them

→ Even though there is almost always at least one, and generally more than one, level of loop *involved* with our optimization target, *loop-specific* optimizations are not always applicable.

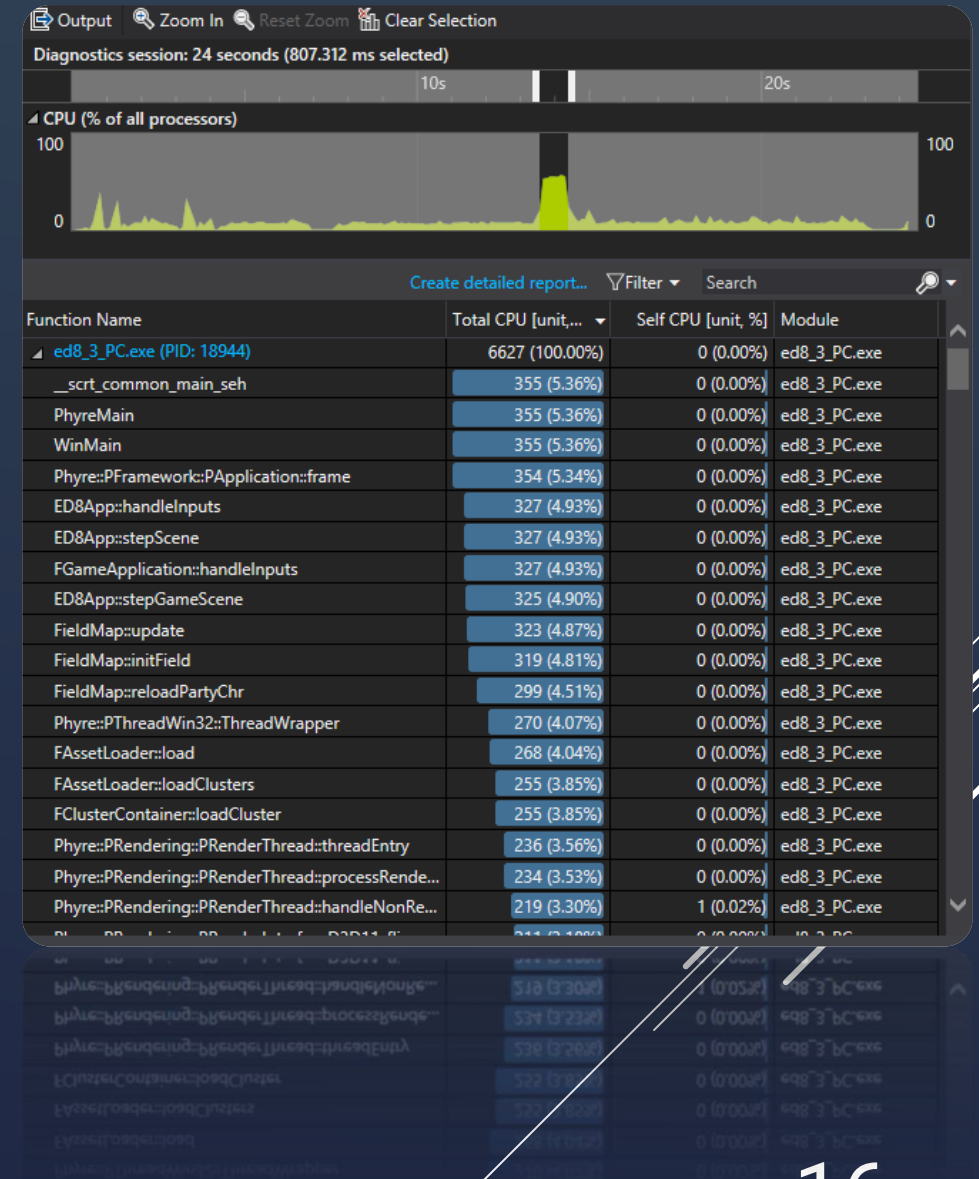
→ We need more **tool support** to find relevant parts of large programs!

PROFILERS

The most important category of tools for performance optimization

PROFILING – OVERVIEW

- ▶ **Profiling** is a specific type of performance measurement, with the primary goal of **characterizing** program behaviour to enable **targeted optimization**
- ▶ Generally at least partially **automated**
- ▶ Since it is a type of performance measurement, the factors influencing the quality of the result which we discussed in the experimentation chapter still apply!
 - ▶ Reproduce the production scenario, variance with input data, ...



TECHNICAL BACKGROUND

- ▶ Details may vary with the programming language and tools used
- ▶ **2 fundamental, popular techniques:**
 - ▶ **Tracing**-based profiling
 - ▶ **Sampling**-based profiling
- ▶ Also combinations of both, and more specialized/rare types
 - discussed later

General challenge: **Balance** between exact and detailed measurements and low **Perturbation** (influence of profiling itself on the results)

TRACING-BASED PROFILING

- ▶ **Trace:** recording of events
- ▶ Requires *tracing code* to be inserted
 - ▶ Either manually or automated
 - ▶ Referred to as *instrumentation*
- ▶ **Advantages:** fine-grained measurements; target selection
- ▶ **Disadvantages:** overhead/perturbation; potential manual effort; might not know target

```
...  
call f_x  
...  
...  
call f_y  
...  
f_x:  
...  
ret  
f_y:  
...  
ret
```

Instrumentation



```
...  
call f_x  
...  
...  
call f_y  
...  
f_x:  
log_ev(start, f_x);  
...  
log_ev(end, f_x);  
ret  
f_y:  
log_ev(start, f_y);  
...  
log_ev(end, f_y);  
ret
```

SAMPLING-BASED PROFILING

- ▶ *Idea*: **periodically** interrupt running program, **record** currently executing functions
- ▶ Usually based on **stack inspection**, and existing **debug information**
- ▶ **Advantages:**
 - ▶ Automated (available for most serious languages)
 - ▶ Easily adjust detail <-> perturbation tradeoff (by setting **sampling frequency**)
- ▶ **Disadvantages:**
 - ▶ Timelines and dependencies might be difficult to understand
 - ▶ Can not analyse short-term spikes

REPRESENTING PROFILING RESULTS

- ▶ Profiling, especially sampling-based, might generate very **large amounts of data**
- ▶ Meaningfully **representing** this data can be an important challenge
 - ▶ Remember, we want to use it to guide our optimization targets and decisions!
- ▶ Many options, I'll just present a small subset now

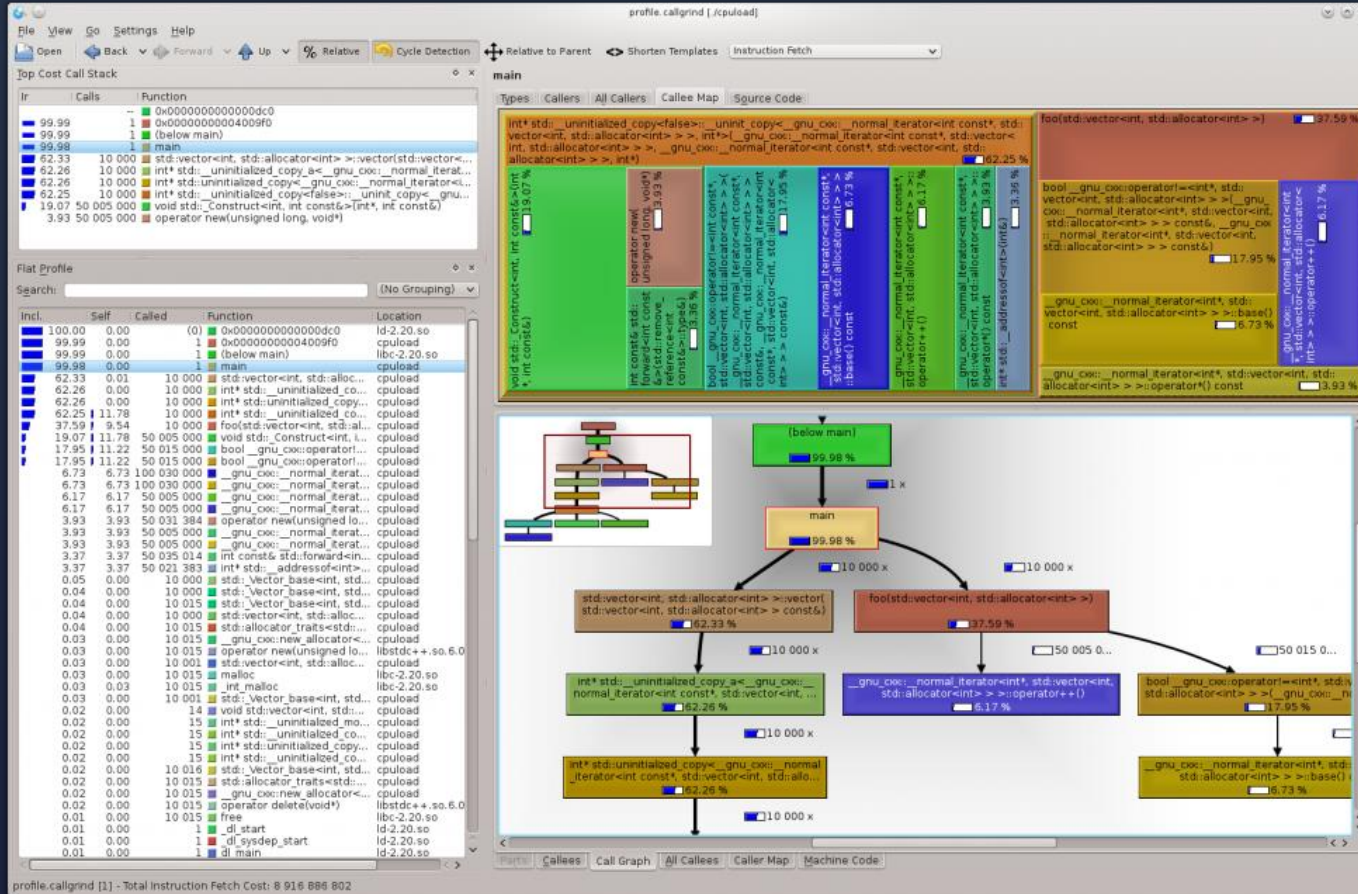
CLASSIC “FLAT PROFILE”

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
48.76	17.67	17.67	170	0.10	0.10	resid
27.85	27.76	10.09	168	0.06	0.06	psinv
7.01	30.30	2.54	147	0.02	0.02	interp
6.10	32.51	2.21	147	0.02	0.02	rprj3
5.08	34.35	1.84	131072	0.00	0.00	vranlc
2.40	35.22	0.87	151	0.01	0.01	zero3
1.16	35.64	0.42	4	0.11	0.11	norm2u3
1.08	36.03	0.39	2	0.20	1.12	zran3
0.58	36.24	0.21	487	0.00	0.00	comm3
0.00	36.24	0.00	131642	0.00	0.00	randlc
0.00	36.24	0.00	608	0.00	0.00	bubble
0.00	36.24	0.00	21	0.00	1.48	mg3P
0.00	36.24	0.00	6	0.00	0.00	power
0.00	36.24	0.00	4	0.00	0.00	elapsed_time
0.00	36.24	0.00	4	0.00	0.00	wtime_
0.00	36.24	0.00	2	0.00	0.00	setup
0.00	36.24	0.00	2	0.00	0.00	timer_clear
0.00	36.24	0.00	2	0.00	0.00	timer_read

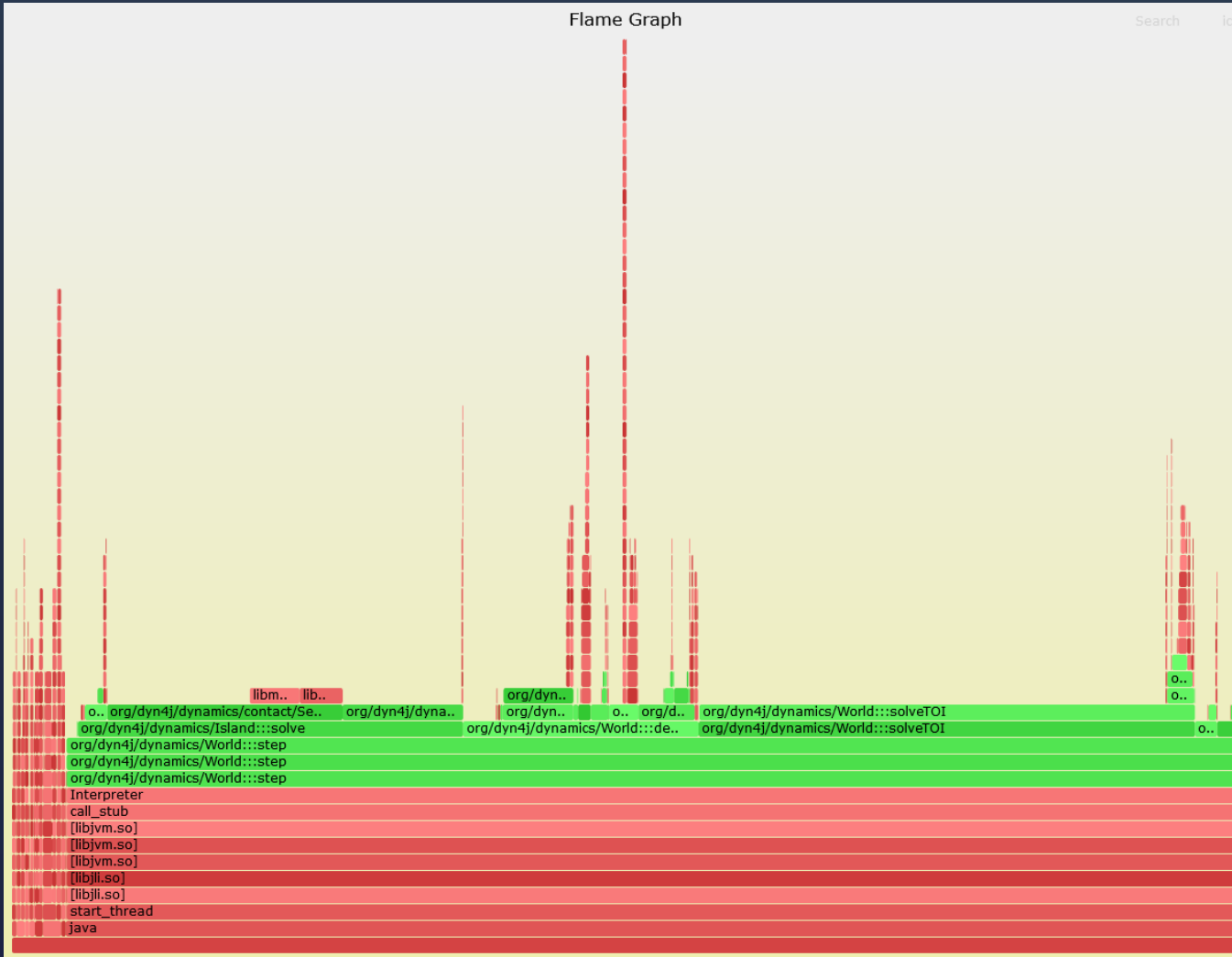
- ▶ This sample was generated by **gprof (GNU profiler)**
- ▶ Available in similar formats for all relevant platforms
- ▶ Allows basic investigation of **performance hot spots**

CALL GRAPH / CALLEE MAP



- ▶ This sample was generated by **Valgrind / Callgrind**
- ▶ Graphical representation sometimes useful to see overall relationship between function
- ▶ Gain overview of CPU time spent

FLAME GRAPH



- ▶ Generated by **perf** with the FlameGraph scriptset
- ▶ Fast navigation and overview by more directly associating **call stacks** with **time**
- ▶ In this special case: additionally using **perf-map-agent** to integrate data from hosted JIT languages

ADVANCED PROFILING

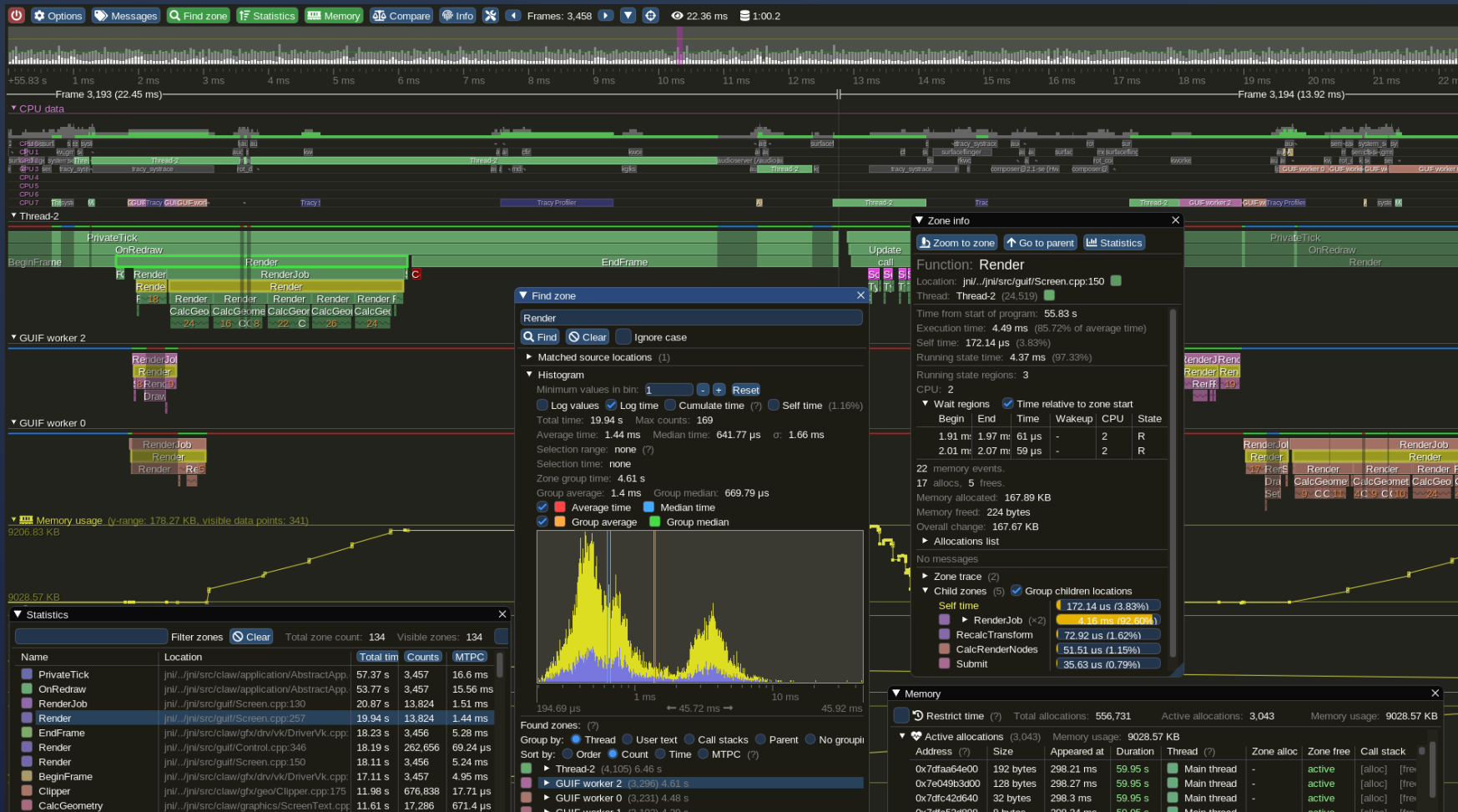
- ▶ Beyond traditional sampling-based and trace-based profiling, there are some more advanced / newer options:
 - ▶ **Hybrid Profiling** – attempting to combine the advantages of both
 - ▶ **Causal Profiling** – attempts to find the parts of code for which optimization is most impactful, even for asynchronous or parallel programs

HYBRID PROFILING

- ▶ Information on the left generated by the Tracy profiler

- ▶ Other examples: Optick, Superliminal

- ▶ Shows sampling information **inside** a timeline of traced events



25

CAUSAL PROFILING

For parallel, or asynchronous, ... programs:



- ▶ *Optimizing functions with high CPU-% might not have the impact you expect!*
 - ▶ Not on the critical path of execution, or time spent in locks, ...

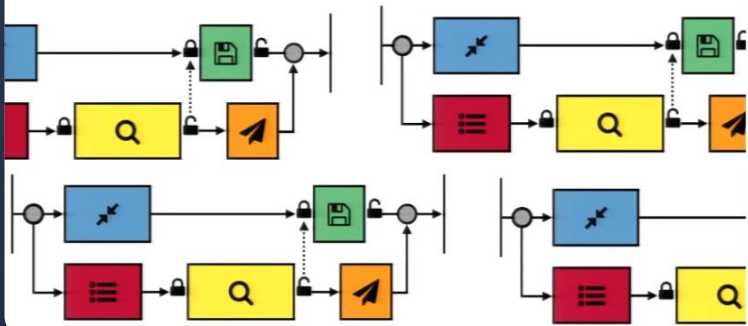
→ **Causal** profiling works by

1. Setting a **progress metric**
2. Measuring how **slowdown** in specific functions affects that progress


<https://www.youtube.com/watch?v=jE0V-p1odPg>

Progress Points

One progress point measures throughput.
If I speed up , how much faster do I run ?



SOSP'15



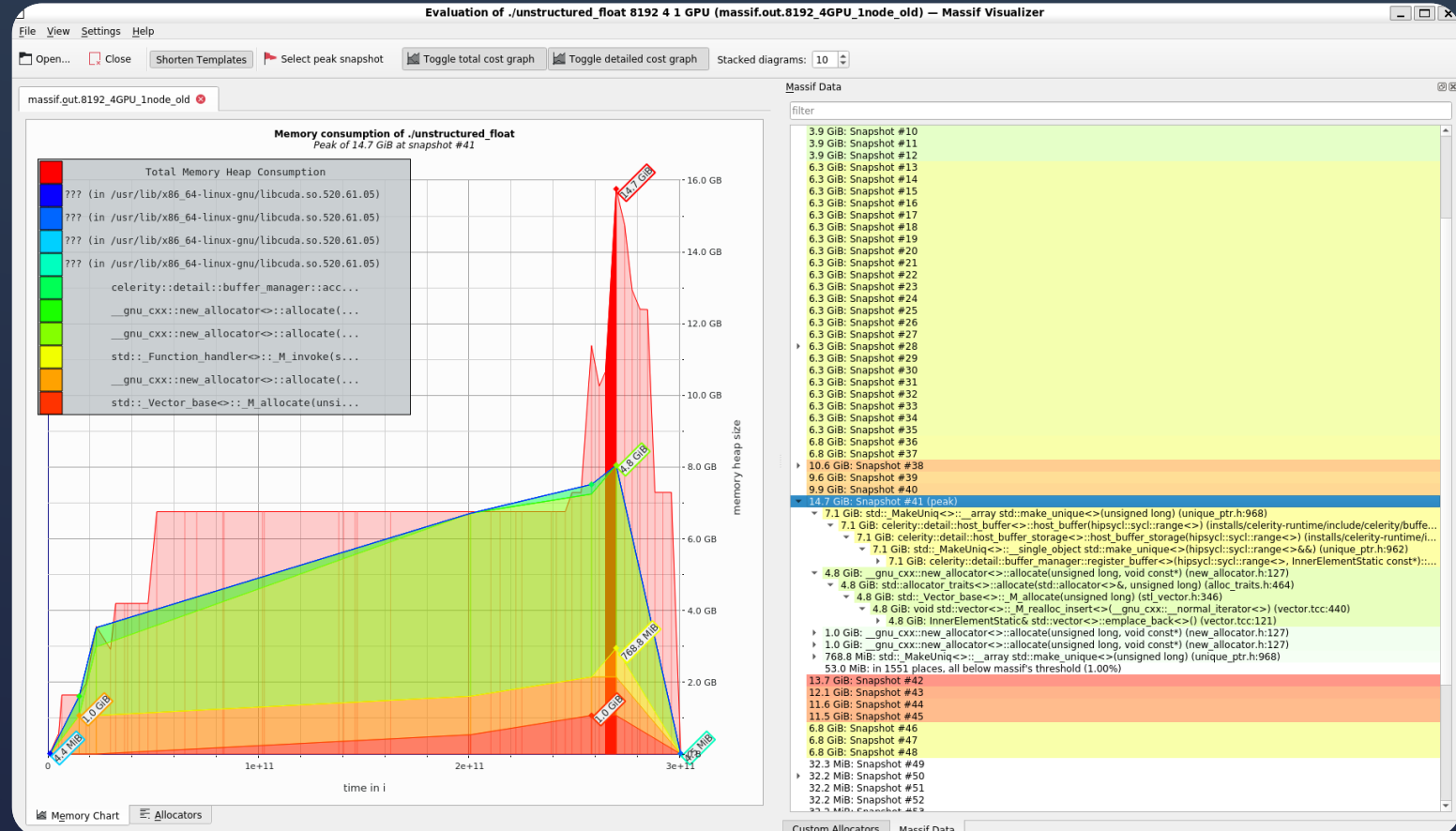


THE LAND BEFORE TIME

What else can we measure?

MEMORY CONSUMPTION

- ▶ Heap profilers:
Massif tool in Valgrind
 - ▶ Use e.g. with
Massif-Visualizer KDE tool
- ▶ Can provide detailed information on **when** and **where** allocations happen
- ▶ Note: potentially *massive* execution time perturbation



OTHER OPTIONS FOR MEMORY

- ▶ When memory management is a significant issue in your application, you might benefit from using a **high-performance memory allocator**
 - ▶ More on this later
- ▶ Important for now: that allocator might support at least some basic level of heap profiling with **less overhead** than tools like Massif

```
> env MIMALLOC_SHOW_STATS=1 ./cfrac 175451865205073170563711388363
```

```
175451865205073170563711388363 = 374456281610909315237213 * 468551
```

heap stats:	peak	total	freed	unit	
normal 2:	16.4 kb	17.5 mb	17.5 mb	16 b	ok
normal 3:	16.3 kb	15.2 mb	15.2 mb	24 b	ok
normal 4:	64 b	4.6 kb	4.6 kb	32 b	ok
normal 5:	80 b	118.4 kb	118.4 kb	40 b	ok
normal 6:	48 b	48 b	48 b	48 b	ok
normal 17:	960 b	960 b	960 b	320 b	ok

heap stats:	peak	total	freed	unit	
normal:	33.9 kb	32.8 mb	32.8 mb	1 b	ok
huge:	0 b	0 b	0 b	1 b	ok
total:	33.9 kb	32.8 mb	32.8 mb	1 b	ok

malloc requested: 32.8 mb

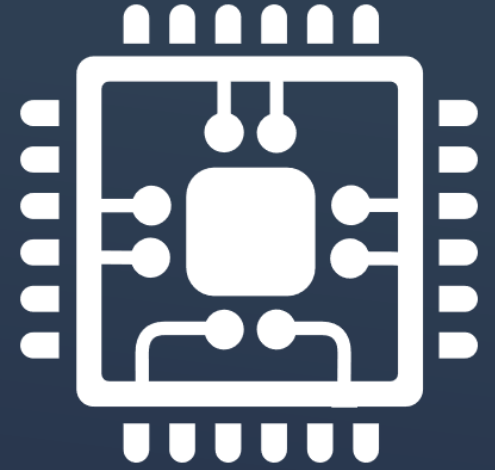
committed:	58.2 kb	58.2 kb	58.2 kb	1 b	ok
reserved:	2.0 mb	2.0 mb	2.0 mb	1 b	ok
reset:	0 b	0 b	0 b	1 b	ok
segments:	1	1	1		
-abandoned:	0				
pages:	6	6	6		
-abandoned:	0				
mmaps:	3				
mmap fast:	0				
mmap slow:	1				
threads:	0				
elapsed:	2.022s				
process:	user: 1.781s, system: 0.016s, faults: 756, reclaims: 0, rss: 2.7 mb				

POWER AND ENERGY

- ▶ Power measurements are usually quite hardware-specific
- ▶ For Intel CPUs, the Running Average Power Limit (**RAPL**) energy reporting facilities provide **estimates** of the power consumption at a core and CPU package level
 - ▶ Since it's an estimate, the temporal resolution is quite good
 - ▶ RAPL registers are also exposed by newer AMD CPUs
 - ▶ No generalized facilities across ARM CPUs that I'm aware of



CPU COUNTERS



- ▶ **CPU Counters** track information about various factors that might influence program performance
 - ▶ The **types of instructions** being executed
 - ▶ The effectiveness of various **levels of caching** for a given workload
 - ▶ Other factors with a performance impact, such as **branch (mis-)prediction** or **TLB hit rates**
- ▶ Generally **CPU-specific**, but tools exist to slightly simplify cross-HW usage

THE PERF TOOL

- ▶ On Linux systems, the **perf** tool supports various CPU counter measurements
 - ▶ Provides **virtual events** for common features, which map to the correct HW implementation
 - ▶ HW-specific events are also accessible, but need to find those in e.g. CPU manufacturer spec sheets

```
> perf list

branch-instructions OR branches    [Hardware event]
branch-misses                      [Hardware event]
[...]
instructions                       [Hardware event]
ref-cycles                        [Hardware event]

alignment-faults                  [Software event]
bpf-output                        [Software event]
context-switches OR cs            [Software event]
[...]
page-faults OR faults             [Software event]
task-clock                        [Software event]

L1-dcache-load-misses             [Hardware cache event]
[...]
L1-icache-loads                   [Hardware cache event]
LLC-load-misses                   [Hardware cache event]
[...]
LLC-stores                        [Hardware cache event]
branch-load-misses                [Hardware cache event]
branch-loads                      [Hardware cache event]
dTLB-load-misses                  [Hardware cache event]
[...]
iTLB-loads                        [Hardware cache event]

branch-instructions OR cpu/branch-ins. [Kernel PMU event]
[...]
ref-cycles OR cpu/ref-cycles/      [Kernel PMU event]
[...]
```


CPU COUNTER CHALLENGES

- ▶ CPU counters are a very low-level instrument
 - ▶ Merely *existing* has some cost in terms of HW
- The number of CPU counters that can be **active at the same time** is **limited**
 - There might also be additional constraints on groups of counters etc.
- One option is to collect measurements across several runs, but of course all the constraints on accuracy etc. we discussed previously apply

CONCLUSION

SUMMARY

► Interpretation

- General idea & bottlenecks
- Roofline model
- Dynamic nature of the optimization process

► Loops & basic runtime estimates

► Profilers

- Technical background
- Tracing & sampling
- Profiler & visualization examples
- Hybrid profiling
- Causal profiling

► Beyond Time

- Memory profiling
- Power estimates
- CPU counters & perf

QUESTIONS ?

