

# Profiling using Performance Counters

Luca Tornatore - I.N.A.F.

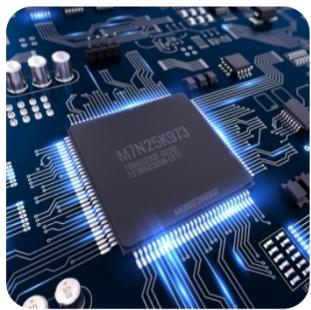


**“Foundation of HPC” course**



DATA SCIENCE &  
SCIENTIFIC COMPUTING  
2021-2022 @ Università di Trieste

# Outline



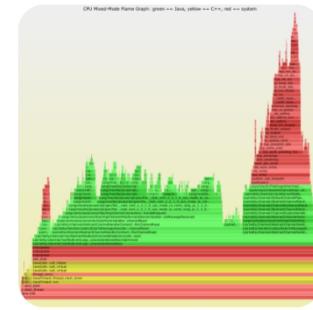
PMUs



What is  
perf



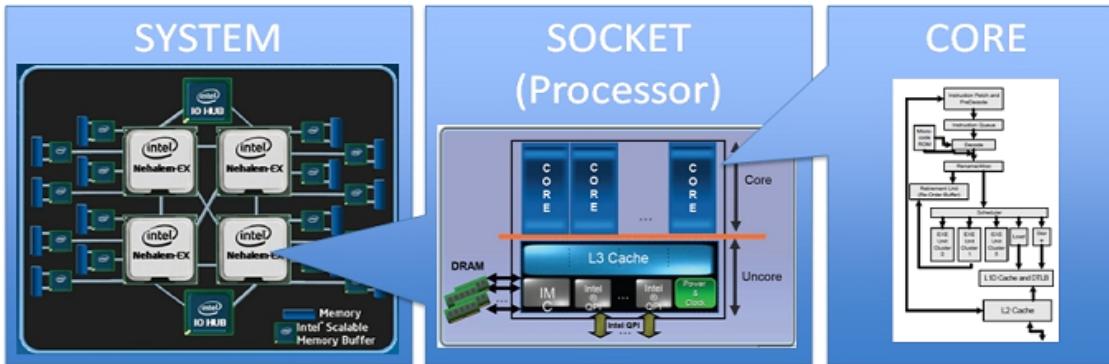
Using  
perf



Profiling  
with perf



# How well does the CPU work ?



The CPU utilization number – the figure that the \*NIX "top" utility reports – is the **fraction of time slots** that the CPU scheduler in the OS assigned to **execution** of running programs or the OS itself.

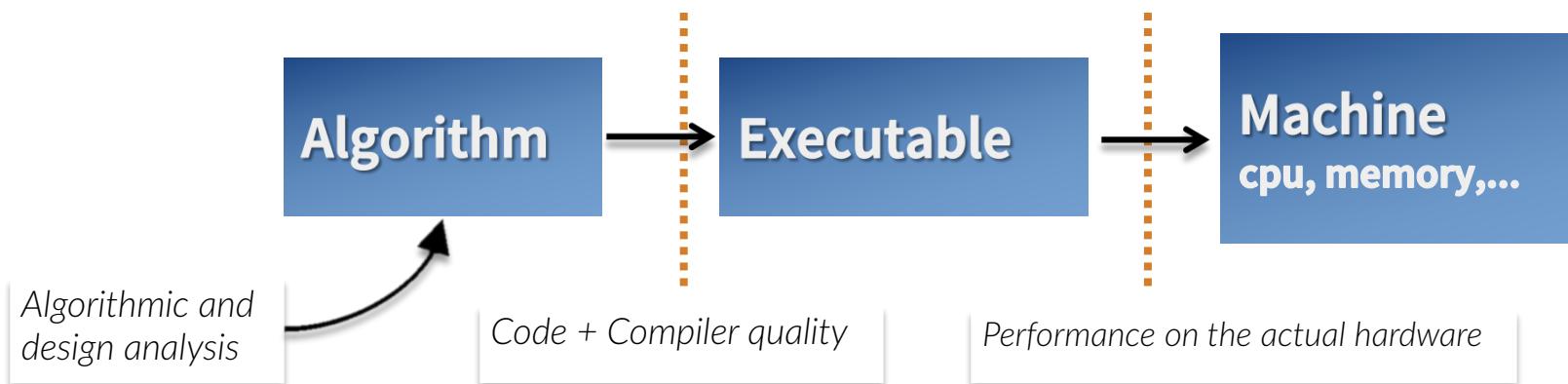
For compute-bound workloads, this figure predicted the remaining CPU capacity fairly well for architectures of 80ies and early 90ies.

This metric is now **unreliable** because of introduction of multi core and multi CPU systems, multi-level caches, non-uniform memory, simultaneous multithreading (SMT), pipelining, out-of-order execution.



PMUs

# | Who makes the CPU working well?



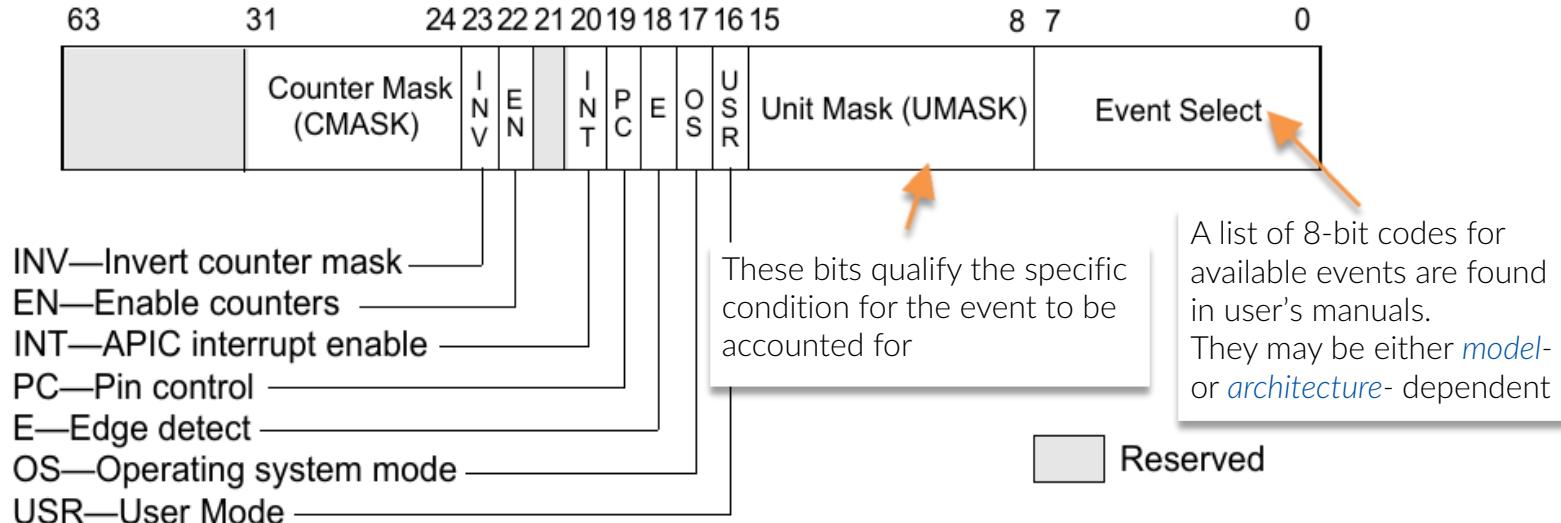
Modern high-end processors provide on-chip hardware that monitors processor's performance and inside events.

That allows to obtain a “precise” and dynamic picture of CPU resources utilization.

These data can guide performance improvement efforts



# Performance Monitoring Units



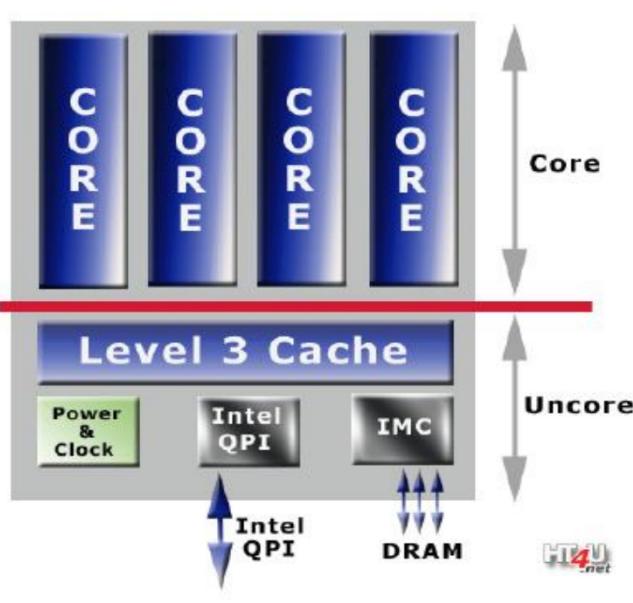
The PMU consists of two types of registers:

- Performance Event Select Register
- Performance Monitor Counter



PMUs

# | Type of intel® counters

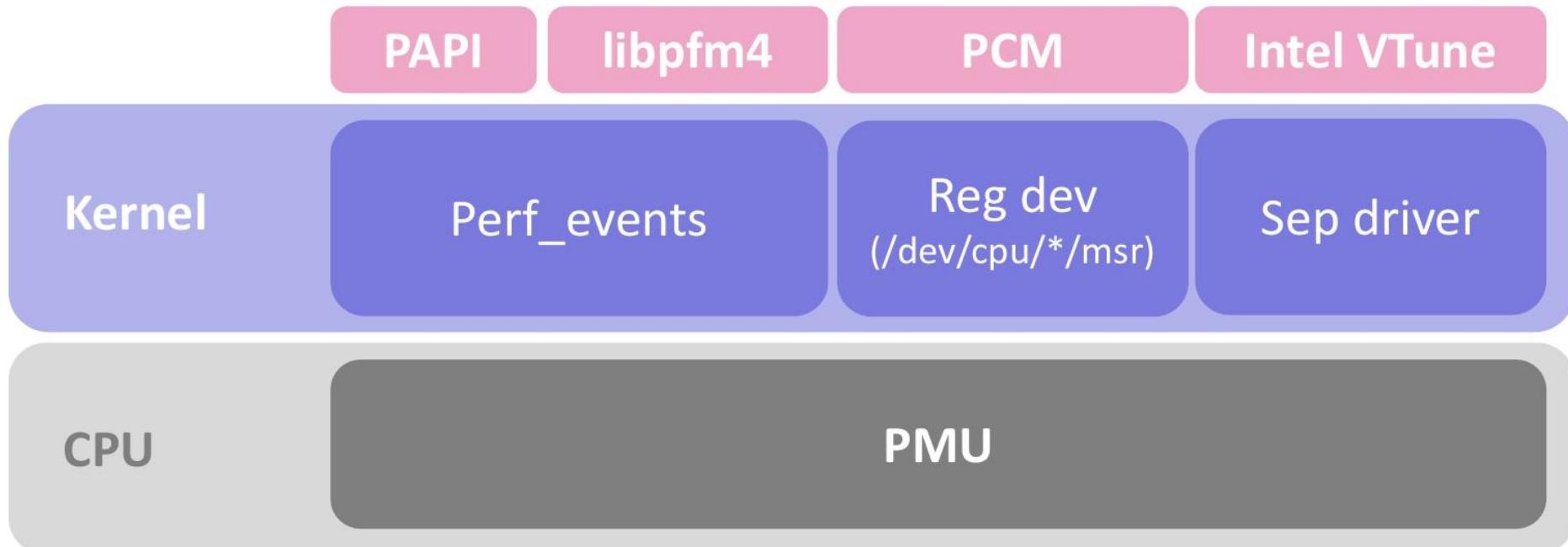


- **Fixed function counters**
  - Predefined events that are commonly used
  - TSC, instructions retired, core clock cycles, ...
- **General purpose performance counters**
  - can be programmed to follow a specific event
- **Precise Event-Based Sampling (PEBS)**
  - Can keep track of architectural state right after instruction causes event
  - Can trigger interrupt (PMI) coupled to counter



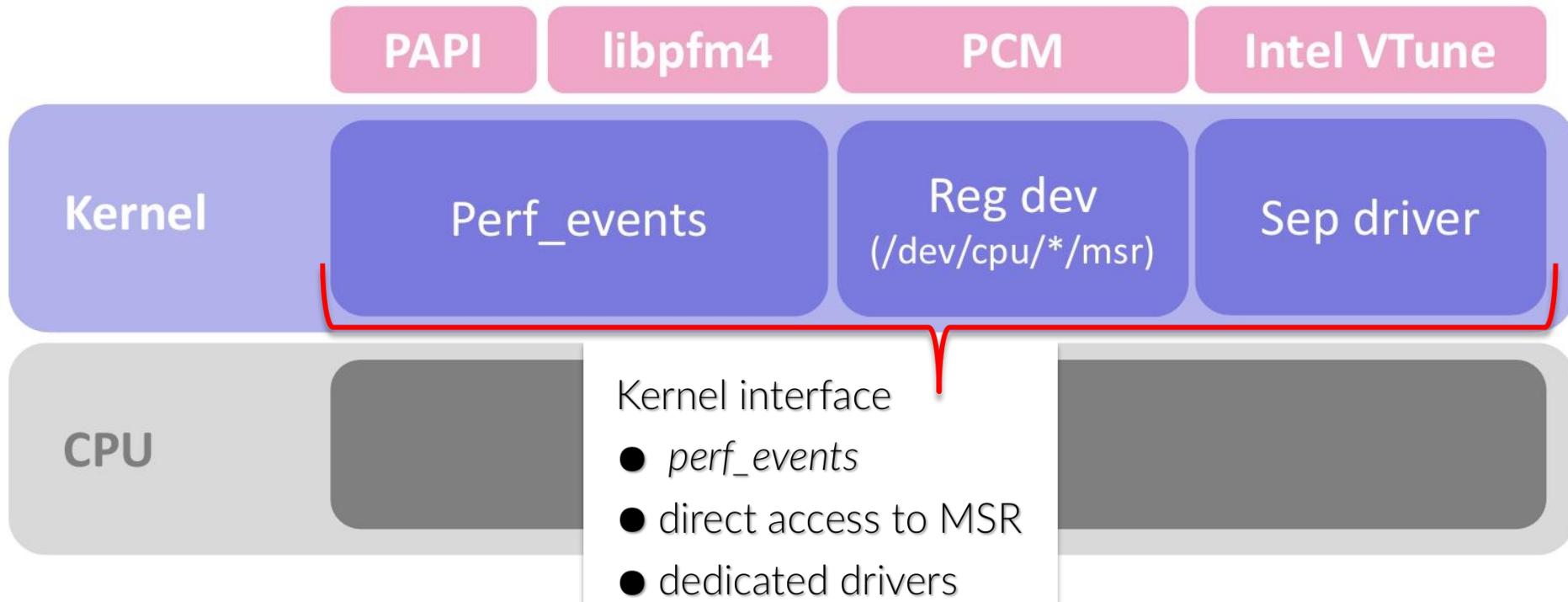
PMUs

# | Accessing the counters



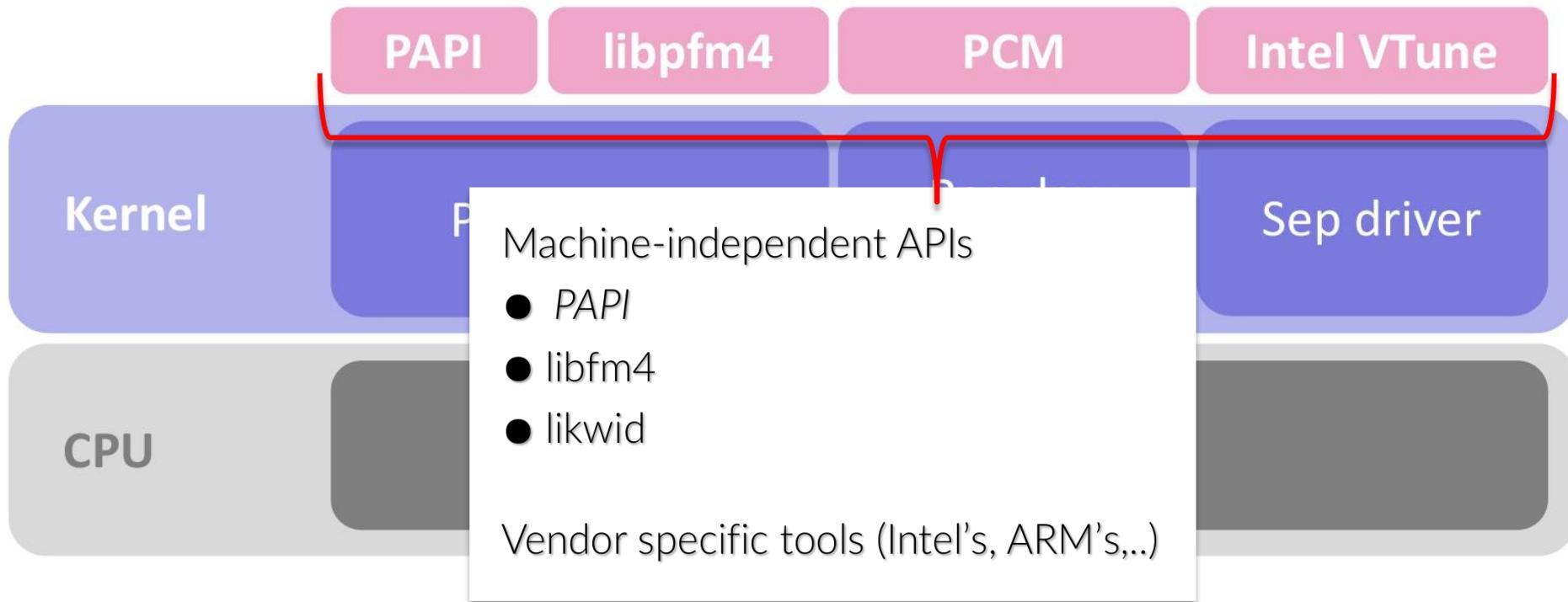


# | Accessing the counters





# | Accessing the counters





Direct acces to  
fixed-function  
perf. count.  
through `rdpmc`

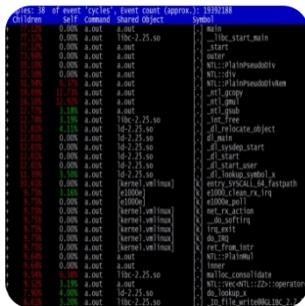
Using PERF, PAPI, PCM,  
LIKWID, ..., is  
definitely easier

```
01 // rdpmc_instructions uses a "fixed-function" performance counter to return the count
02 // the current core in the low-order 48 bits of an unsigned 64-bit integer.
03 unsigned long rdpmc_instructions()
04 {
05     unsigned a, d, c;
06
07     c = (1<<30);
08     __asm__ volatile("rdpmc" : "=a" (a), "=d" (d) : "c" (c));
09
10     return ((unsigned long)a) | (((unsigned long)d) << 32);
11 }
12
13 // rdpmc_actual_cycles uses a "fixed-function" performance counter to return the count
14 // executed by the current core. Core cycles are not accumulated while the processor
15 // which is used when the operating system has no task(s) to run on a processor
16 unsigned long rdpmc_actual_cycles()
17 {
18     unsigned a, d, c;
19
20     c = (1<<30)+1;
21     __asm__ volatile("rdpmc" : "=a" (a), "=d" (d) : "c" (c));
22
23     return ((unsigned long)a) | (((unsigned long)d) << 32);
24 }
25
26 // rdpmc_reference_cycles uses a "fixed-function" performance counter to return the count
27 // CPU core cycles executed by the current core. This counts at the same rate as
28 // when the core is in the "HALT" state. If a timed section of code shows a large
29 // rdpmc_reference_cycles, the processor probably spent some time in a HALT state.
30 unsigned long rdpmc_reference_cycles()
31 {
32     unsigned a, d, c;
33
34     c = (1<<30)+2;
35     __asm__ volatile("rdpmc" : "=a" (a), "=d" (d) : "c" (c));
36
37     return ((unsigned long)a) | (((unsigned long)d) << 32);
38 }
```

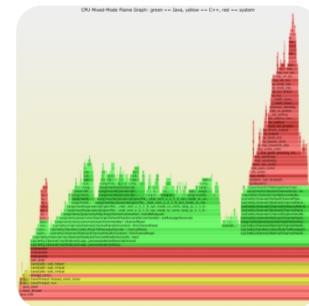
## Outline



# What is perf



# Using perf



# Profiling with perf



# What is perf?

PERF – Performance Events for Linux – is the standard profiling infrastructure on Linux, built in the kernel since version 2.6.

It strips away the CPUs hardware differences among different systems and presents an abstracted unique command-line interface for performance measurements.

PERF is the standard way to access the hardware performance counters, in both sampling and counting mode.

It consists of:

- a kernel SYSCALL that provide access to both system-software and hardware events;
- a collection of user-space tools to collect, display and analyze performance data.

References:

- <https://perf.wiki.kernel.org/index.php/Tutorial>
- <http://brendangregg.com/perf.html>



What is  
perf?

# Hello perf

```
luca@GGG:~% perf
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
annotate          Read perf.data (created by perf record) and display annotated code
archive           Create archive with object files with build-ids found in perf.data file
bench              General framework for benchmark suites
buildid-cache     Manage build-id cache.
buildid-list      List the buildids in a perf.data file
c2c               Shared Data C2C/HITM Analyzer.
config             Get and set variables in a configuration file.
data               Data file related processing
diff               Read perf.data files and display the differential profile
evlist             List the event names in a perf.data file
ftrace             simple wrapper for kernel's ftrace functionality
inject             Filter to augment the events stream with additional information
kallsyms          Searches running kernel for symbols
kmem               Tool to trace/measure kernel memory properties
kvm                Tool to trace/measure kvm guest os
list               List all symbolic event types
lock               Analyze lock events
mem                Profile memory accesses
record              Run a command and record its profile into perf.data
report              Read perf.data (created by perf record) and display the profile
sched               Tool to trace/measure scheduler properties (latencies)
script              Read perf.data (created by perf record) and display trace output
stat                Run a command and gather performance counter statistics
test                Runs sanity tests.
timechart          Tool to visualize total system behavior during a workload
top                 System profiling tool.
probe               Define new dynamic tracepoints
trace               strace inspired tool
```



What is  
perf?

# Hello perf

```
luca@GGG:~% perf
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

The most commonly used perf commands are:
  annotate          Read perf.data (created by perf record) and display annotated code
  archive           Create archive with object files with build-ids found in perf.data file
  bench              General framework for benchmark suites
  buildid-cache     Manage build-id cache.
  buildid-list      List the buildids in a perf.data file
  c2c               Shared Data C2C/HITM Analyzer.
  config             Get and set variables in a configuration file.
  data               Data file related processing
  diff               Read perf.data files and display the differential profile
  evlist             List the event names in a perf.data file
  ftrace             simple wrapper for kernel's ftrace functionality
  inject             Filter to augment the events stream with additional information
  kallsyms          Searches running kernel for symbols
  kmem               Tool to trace/measure kernel memory properties
  kvm                Tool to trace/measure kvm guest os
  list               List all symbolic event types
  lock               Analyze lock events
  mem                Profile memory accesses
  record              Run a command and record its profile into perf.data
  report              Read perf.data (created by perf record) and display the profile
  sched              Tool to trace/measure scheduler properties (latencies)
  script              Read perf.data (created by perf record) and display trace output
  stat               Run a command and gather performance counter statistics
  test                Runs sanity tests.
  timechart          Tool to visualize total system behavior during a workload
  top                System profiling tool.
  probe              Define new dynamic tracepoints
  trace              strace inspired tool
```

Note: to obtain the list of options for each command use  
**perf command -h**



## TYPE OF EVENTS

The kernel interface can measure and report events from several different sources:

- **software events** : pure software events (ex: context switches)
- **PMU hardware events** : Performance Monitoring Unit (PMU)  
microarchitectural events (ex. Nr. of instructions retired, branch-misses, ... )
- **hardware events** : common hardware events with mnemonic names,  
mapped on actual CPU events
- **tracepoint events** : implemented through the kernel *ftrace* infrastructure.



## HOW MANY EVENTS IN YOUR CPU ?

Let's have a look:

- ▶ perf list
- ▶ showevtinfo [ pmu-tools ]
- ▶ papi\_avail [ PAPI ]
- ▶ likwid-perfctr -e
- ▶ <https://download.01.org/perfmon/>

...hundreds of different events



What is  
perf?

# Names for **perf** Events

Some commonly used events have mnemonic names:

List of pre-defined events (to be used in -e):

branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
bus-cycles	[Hardware event]
cache-misses	[Hardware event]
cache-references	[Hardware event]
cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
ref-cycles	[Hardware event]
alignment-faults	[Software event]
bpf-output	[Software event]
context-switches OR cs	[Software event]
cpu-clock	[Software event]
cpu-migrations OR migrations	[Software event]
dummy	[Software event]
emulation-faults	[Software event]
major-faults	[Software event]
minor-faults	[Software event]
page-faults OR faults	[Software event]
task-clock	[Software event]
L1-dcache-load-misses	[Hardware cache event]
L1-dcache-loads	[Hardware cache event]
L1-dcache-stores	[Hardware cache event]
L1-icache-load-misses	[Hardware cache event]
LLC-load-misses	[Hardware cache event]
LLC-loads	[Hardware cache event]
LLC-store-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]
dTLB-loads	[Hardware cache event]
dTLB-store-misses	[Hardware cache event]
dTLB-stores	[Hardware cache event]



An event can also be specified by its processor-specific identifier, that usually is a hex value

Example:

```
"EventCode": "0x00",
  "UMask": "0x01",
  "EventName": "INST_RETIRED.ANY",
  "BriefDescription": "Instructions retired from execution."
```

```
"EventCode": "0x00",
  "UMask": "0x02",
  "EventName": "CPU_CLK_UNHALTED.THREAD",
  "BriefDescription": "Core cycles when the thread is not in halt state"
```



You can specify events by their raw code if not present in `perf list`

```
perf stat -e r5100c0 ...
```

You may obtain the hex code in several way

1. Look in the list provided by Intel, combining “Umask” and “Event Code” fields

```
perf stat -e r<umask><eventselector> ...
perf stat -e cpu/event=0xcode,umask=0xcode/u ...
```

1. Use `libpfm4`, using `showevtinfo` and then `check_events`:

```
check_events <event name>:<umask>[(:modifiers)*]
```

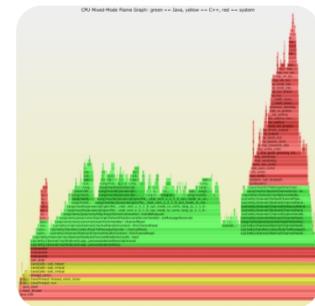
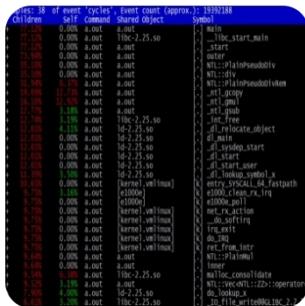
## Outline



PMW



# What is perf



# Using perf

# Profiling with perf



- Which code segments take most of the **execution time** ?  
What are the **causes** ? (*cache misses, branch misses, non-optimal pipelines..*)
- Do the performance counters **suggest** something about performance issues ?
- Do the performance counters offer some hints on how to **fix** the issues?

Workflow:

1. **Run** the program and **collect** as many data as possible
2. **Analyse** the collected data and **profile** the program



## EVENT-based profiling

can be accessed from inside the code  
collecting precise values of events  
counters in a given time range



## SAMPLE-based profiling

periodic sampling of program/system  
status and of event counters values



## EVENT-based profiling

can be accessed from inside the code  
collecting precise values of events  
counters in a given time range



# Counting events with `perf`

For all supported events, `perf stat` can count during a process execution:

```
luca@GGG:~% perf stat -B dd if=/dev/zero of=/dev/null count=10000000
10000000+0 records in
10000000+0 records out
512000000 bytes (512 MB, 488 MiB) copied, 1.0796 s, 474 MB/s

Performance counter stats for 'dd if=/dev/zero of=/dev/null count=10000000':
      1078.764601      task-clock (msec)          #      0.997 CPUs utilized
                  9      context-switches        #      0.008 K/sec
                  1      cpu-migrations         #      0.001 K/sec
                 74      page-faults           #      0.069 K/sec
  2,152,108,887      cycles                #      1.995 GHz
  1,986,849,204      instructions          #      0.92  insn per cycle
  387,532,518       branches              #   359.237 M/sec
     8,022,754      branch-misses        #      2.07% of all branches

  1.081948899 seconds time elapsed
```



# Counting events with **perf**

You can specify the events you want to profile by their mnemonic names:

```
perf stat -e cycles:.,instructions:.,branch-misses:.,cache-misses:..
```

If you specify more events than available counters, the kernel uses time multiplexing to sample all the events; then perf scales the count based on running time vs the total amount of time that a given event has been active

```
final_count = raw_count * time_enabled / time_running
```



# Counting events with `perf`

Processor-wide mode:

```
perf stat -e cycles:u -a ./executable arguments
```

Sampling a subset of cores:

```
perf stat -e cycles:u -a -C 1,4-5 ./executable arguments
```

Profile a given process:

```
perf stat -e cycles:u -p PID sleep 2
```

Pretty printing of large numbers

```
perf stat -B -e cycles:u -e instructions:u -e branch-misses:u ./executable arguments
```

Repeat and average measures

```
perf stat -r 10 -B -e cycles:u -e instructions:u ./executable arguments
```



## SAMPLE-based data collection

periodic sampling of program/system status and of event counters values



# Collecting information with `perf`

You can collect much more information, with line-level detail:

```
perf record -e cycles:.,instructions:.,branch-misses:.,cache-misses:.
```

A file named `perf.data` will be produced, with all the data collected during the `perf` activity.

At odds with `perf stat`, this mode profiles CPU usage based on sampling the instruction pointer or stack tree at a given rate:

```
perf record -F fff -a -k CLOCK_MONOTONIC -- sleep 10
```

Sample the whole systems (`-a`, all CPUs) at fff Hertz (`-F fff`) without accounting for the perf activity (`--`) for 10 secs.



# Sampling the call stack

You can collect dynamic information about the call-stack  
(so that to obtain the call graph)

```
perf record --call-graph <record_mode [,record_size ]>
```

Where `record_mode` can be `<fp|dwarf|lbr>` and `record_size` is the max size of stack recording (in bytes) for `dwarf` mode (def: 8192)

You can specify `-a` (all CPUs) ; `-c`, `--cpu <cpu>` ; `-s`, `--per-thread` ;

Now, let's go back to our friend `lie.c` to check whether `perf` is mislead as much as `gprof` was...



# Sampling the call stack

```
perf report --call-graph graph,0.6,100,caller --stdio --no-children
```

```
# Total Lost Samples: 0
#
# Samples: 740  of event cycles:ppp
# Event count (approx.): 556806058
#
# Overhead  Command  Shared Object      Symbol
# .....  .....
#
99.61%  lie1.g    lie1.g          [.] loop
|
---_start
    __libc_start_main
    main
    |
    --98.73%--heavy
    |         loop
    |
    --0.88%--light
    |         loop
```

caller order

```
# Total Lost Samples: 0
#
# Samples: 740  of event cycles:ppp
# Event count (approx.): 556806058
#
# Overhead  Command  Shared Object      Symbol
# .....  .....
#
99.61%  lie1.g    lie1.g          [.] loop
|
---loop
|
---98.73%--heavy
    main
    __libc_start_main
    _start
|
---0.88%--light
    main
    __libc_start_main
```

callee order



# | Collecting information with `perf`

As well, you can record performance events along with the call stack

```
perf record --call-graph ... -e event_1:.,event_2:.,event_3:.. ...
```

With all the same options valid before.

Check `perf record -h` for more details.

Note:

remember to compile (with `gcc`) with  
**-fno-omit-frame-pointer**  
in order to reconstruct the call stack

# Outline



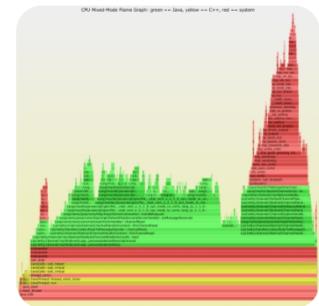
PMUs



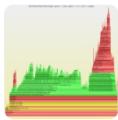
What is  
perf



Using  
perf



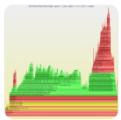
Profiling  
with perf



Once you have `perf.data` file,  
you can easily get the call graph

```
perf report -call-graph --stdio
```

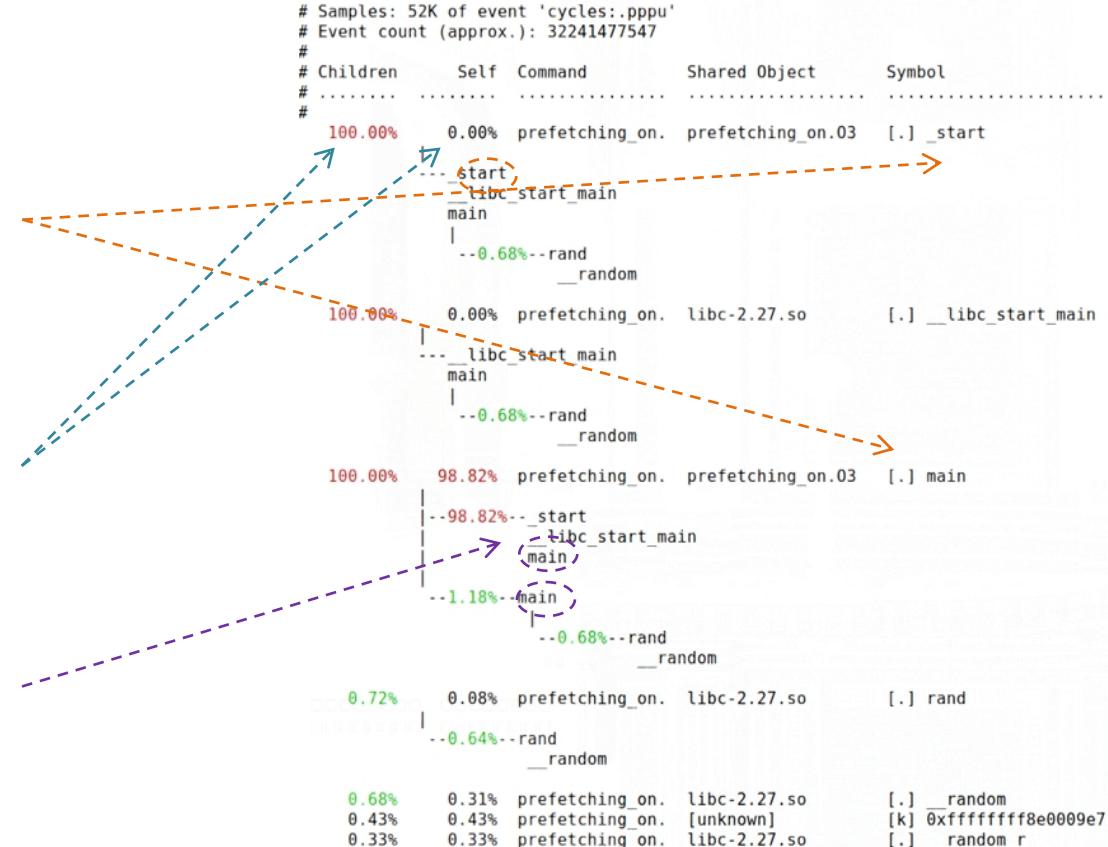
```
# Samples: 52K of event 'cycles::pppu'
# Event count (approx.): 32241477547
#
# Children      Self   Command           Shared Object       Symbol
# .....          .....
#
# 100.00%    0.00%  prefetching_on.  prefetching_on.03  [.] _start
|   ...
|   ---_start
|   ---_libc_start_main
|   main
|   |
|   ---0.68%--rand
|       __random
|
100.00%    0.00%  prefetching_on.  libc-2.27.so        [.] __libc_start_main
|   ...
|   ---_libc_start_main
|   main
|   |
|   ---0.68%--rand
|       __random
|
100.00%    98.82%  prefetching_on.  prefetching_on.03  [.] main
|   ...
|   ---98.82%--_start
|   ---_libc_start_main
|   main
|   |
|   ---1.18%--main
|       |
|       ---0.68%--rand
|           __random
|
0.72%     0.08%  prefetching_on.  libc-2.27.so        [.] rand
|   ...
|   ---0.64%--rand
|       __random
|
0.68%     0.31%  prefetching_on.  libc-2.27.so        [.] __random
0.43%     0.43%  prefetching_on.  [unknown]          [k] 0xffffffff8e0009e7
0.33%     0.33%  prefetching_on.  libc-2.27.so        [.] __random_r
```

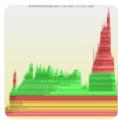


The symbol this call refers to

Children time and self time  
are reported separately

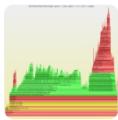
Both *caller* and  
*callee* informations





```
# Samples: 52K of event 'cycles:.pppu'
# Event count (approx.): 32241477547
#
# Overhead  Command          Shared Object      Symbol
# .....  .....
#
#         98.82%  prefetching_on.  prefetching_on.03  [.] main
#                         |           --98.81%--main
#                         |                   _libc_start_main
#                         |                   _start
#
#         0.43%  prefetching_on.  [unknown]          [k] 0xffffffff8e0009e7
#         0.33%  prefetching_on.  libc-2.27.so        [.] __random_r
#         0.31%  prefetching_on.  libc-2.27.so        [.] __random
#         0.08%  prefetching_on.  libc-2.27.so        [.] rand
#         0.03%  prefetching_on.  prefetching_on.03  [.] rand@plt
#         0.00%  prefetching_on.  libc-2.27.so        [.] _dl_addr
#         0.00%  prefetching_on.  ld-2.27.so         [.] _dl_relocate_object
#         0.00%  prefetching_on.  ld-2.27.so         [.] do_lookup_x
#         0.00%  prefetching_on.  ld-2.27.so         [.] strcmp
#         0.00%  prefetching_on.  ld-2.27.so         [.] __GI__tunables_init
```

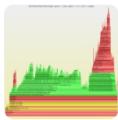
perf report -call-graph -stdio \  
**--no-children**



```
perf report -call-graph -stdio \
--no-children
```

callee format as default

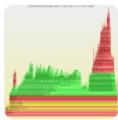
```
# Samples: 52K of event 'cycles:pppu'
# Event count (approx.): 32241477547
#
# Overhead  Command           Shared Object      Symbol
# .....   .....   .....   .....
#
#          98.82%  prefetching_on.  prefetching_on.03  [.] main
#                         |           --98.81%--main
#                         |           _libc_start_main
#                         |           _start
#
#          0.43%  prefetching_on.  [unknown]          [k] 0xffffffff8e0009e7
#          0.33%  prefetching_on.  libc-2.27.so        [.] __random_r
#          0.31%  prefetching_on.  libc-2.27.so        [.] __random
#          0.08%  prefetching_on.  libc-2.27.so        [.] rand
#          0.03%  prefetching_on.  prefetching_on.03  [.] rand@plt
#          0.00%  prefetching_on.  libc-2.27.so        [.] _dl_addr
#          0.00%  prefetching_on.  ld-2.27.so         [.] _dl_relocate_object
#          0.00%  prefetching_on.  ld-2.27.so         [.] do_lookup_x
#          0.00%  prefetching_on.  ld-2.27.so         [.] strcmp
#          0.00%  prefetching_on.  ld-2.27.so         [.] __GI__tunables_init
```



```
perf report --call-graph --stdio \  
--no-children --sort=dso
```

Different ordering are possible:  
comm, dso, symbol,...

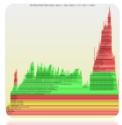
```
# Samples: 52K of event 'cycles:.pppu'  
# Event count (approx.): 32241477547  
#  
# Overhead  Shared Object  
# .....  
#  
98.84%  prefetching_on.03  
|  
---_start  
    __libc_start_main  
    main  
  
0.72%   libc-2.27.so  
|  
---_start  
    __libc_start_main  
    main  
|  
--0.68%--rand  
    __random  
  
0.43%   [unknown]
```



```
perf report --call-graph --stdio \
--no-children --sort=comm
```

Different ordering are possible:  
comm, dso, symbol,...

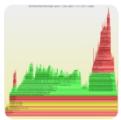
```
# Samples: 52K of event 'cycles:.pppu'
# Event count (approx.): 32241477547
#
# Overhead  Command
# ..... .....
#           100.00% prefetching_on.
|           --100.00%--_start
|           _libc_start_main
|           main
|           --0.68%--rand
|           _random
```



```
perf report --call-graph --stdio \
--no-children
```

```
# Samples: 55K of event 'instructions:u'  
# Event count (approx.): 9679409860  
#  
# Overhead  Command           Shared Object      Symbol  
# .....  
#  
97.01%  prefetching_on.  prefetching_on.03  [.] main  
|  
|--- _start  
|     __libc_start_main  
|     main  
  
2.05%   prefetching_on.  libc-2.27.so        [.] __random  
|  
|--- _start  
|     __libc_start_main  
|     main  
|     rand  
|     random
```

The graph for all the events recorded are shown

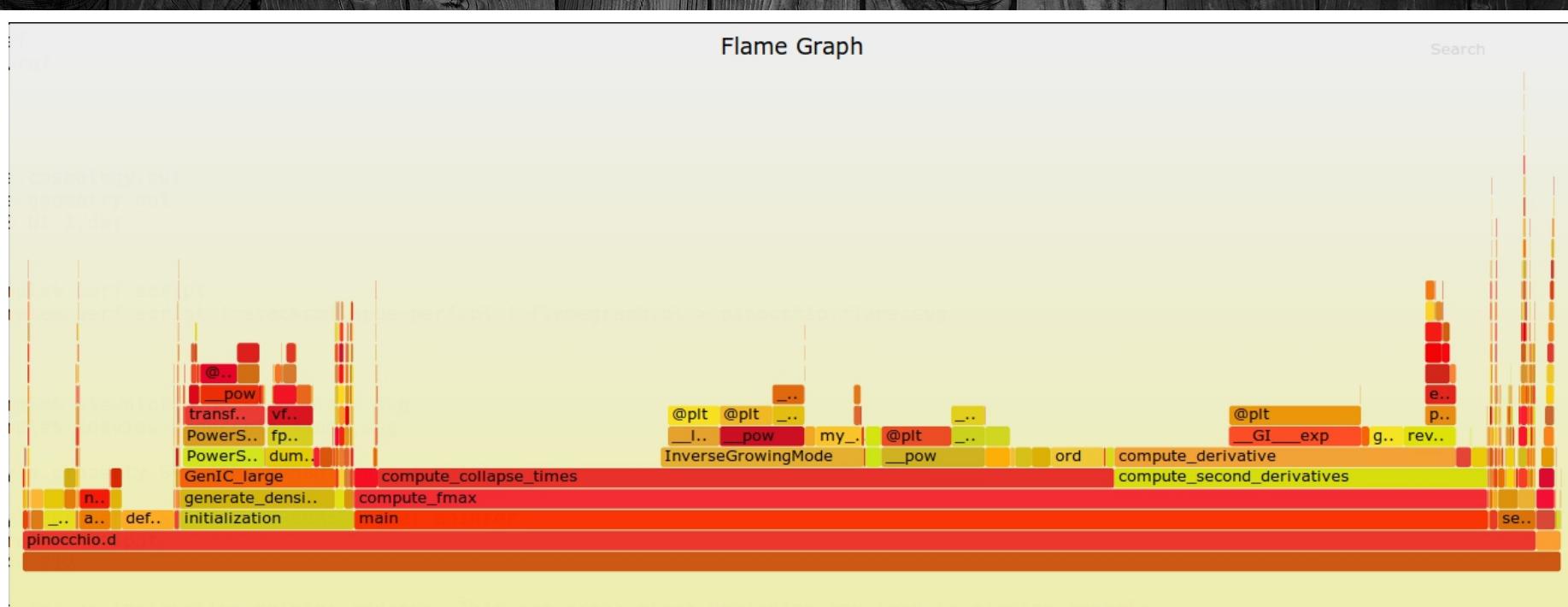
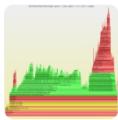


You can browse your code at line-level detail, with reported metrics for all the events that you requested

→ see live examples

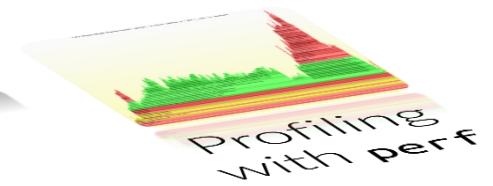
```
main /home/luca/code/HPC LECTURES/prefetching/prefetching on.g03
0.00    shr    $0x1f,%eax
0.09    add    %edx,%eax
0.06    sar    %eax
          mid = (low + high) / 2;
8.90    mov    %eax,%edx
          while(low <= high) {
0.01    cmp    %esi,%ecx
        † jg    23b
          __builtin_prefetch (&data[(low + mid - 1)/2], 0, 1);
15.96  1f7: lea    -0x1(%rcx,%rdx,1),%r10d
0.13    mov    %r10d,%eax
1.06    shr    $0x1f,%eax
0.50    add    %r10d,%eax
          __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 1);
3.68    lea    0x1(%rdx,%r10d
          __builtin_prefetch (&data[(low + mid - 1)/2], 0, 1);
0.10    sar    %eax
          __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 1);
8.42    lea    (%r10,%rsi,1),%r11d
          __builtin_prefetch (&data[(low + mid - 1)/2], 0, 1);
0.12    cltq
0.64    prefetch0 (%r14,%rax,4)
          __builtin_prefetch (&data[(mid + 1 + high)/2], 0, 1);
23.31
0.08    mov    %r11d,%eax
0.11    shr    $0x1f,%eax
0.11    add    %r11d,%eax
0.11    sar    %eax
1.64    movslq %eax,%r11
0.11    prefetch0 (%r14,%r11,4)
          if(data[mid] < Key)
0.92    movslq %edx,%r11
0.03    cmp    (%r14,%r11,4),%edi
0.05    † jle    1e0
          low = mid + 1;
3.61    mov    %r10d,%ecx
          mid = (low + high) / 2;
0.08    mov    %eax,%edx
          while(low <= high) {
0.08    cmp    %esi,%ecx
0.01    † jle    1f7
1.98    23b: add    $0x4,%r15
main():
          for (i = 0; i < Nsearch; i++)
1.75    cmp    %r12,%r15
          † jne    1d0
          found++;
```

Press 'h' for help on key bindings



Have a look at Brend Gregg's blog: <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>

# Outline



## Addendum I

### Elementary intro to PAPI

```
if(Q > 0)
{
    switch(Q)
    {
        case(1): // row y = 0 and/or plane symmetry
            if(subregions[i][BOTTOM][y] == 0)
                // this subregion in quadrant 1 contains the
                // set-up corners for seed sub-region gen
                SBL[y] = 0, SBL[X_] = Nmesh - subregion
                STR[y] = 1, STR[X_] = Nmesh - subregion
                // find horizontal extension in this quad
                Np = STR[X_] - SBL[X_];
                // allocate memory for seeds in this strip
                SEED_y0 = (unsigned int*)malloc(sizeof(unsigned
                int)*Np);
                if(!Internal.mtnic_original_seedtable)
                {
                    // copy original seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_original_seedtable[j];
                }
                else
                    // copy current seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_current_seedtable[j];
            }
        case(2): // row y = 1 and/or plane symmetry
            if(subregions[i][TOP][y] == 0)
                // this subregion in quadrant 2 contains the
                // set-up corners for seed sub-region gen
                SBL[y] = 1, SBL[X_] = Nmesh - subregion
                STR[y] = 0, STR[X_] = Nmesh - subregion
                // find horizontal extension in this quad
                Np = STR[X_] - SBL[X_];
                // allocate memory for seeds in this strip
                SEED_y0 = (unsigned int*)malloc(sizeof(unsigned
                int)*Np);
                if(!Internal.mtnic_original_seedtable)
                {
                    // copy original seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_original_seedtable[j];
                }
                else
                    // copy current seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_current_seedtable[j];
            }
        case(3): // row y = 2 and/or plane symmetry
            if(subregions[i][MIDDLE][y] == 0)
                // this subregion in quadrant 3 contains the
                // set-up corners for seed sub-region gen
                SBL[y] = 2, SBL[X_] = Nmesh - subregion
                STR[y] = 1, STR[X_] = Nmesh - subregion
                // find horizontal extension in this quad
                Np = STR[X_] - SBL[X_];
                // allocate memory for seeds in this strip
                SEED_y0 = (unsigned int*)malloc(sizeof(unsigned
                int)*Np);
                if(!Internal.mtnic_original_seedtable)
                {
                    // copy original seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_original_seedtable[j];
                }
                else
                    // copy current seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_current_seedtable[j];
            }
        case(4): // row y = 3 and/or plane symmetry
            if(subregions[i][TOP][y] == 0)
                // this subregion in quadrant 4 contains the
                // set-up corners for seed sub-region gen
                SBL[y] = 3, SBL[X_] = Nmesh - subregion
                STR[y] = 0, STR[X_] = Nmesh - subregion
                // find horizontal extension in this quad
                Np = STR[X_] - SBL[X_];
                // allocate memory for seeds in this strip
                SEED_y0 = (unsigned int*)malloc(sizeof(unsigned
                int)*Np);
                if(!Internal.mtnic_original_seedtable)
                {
                    // copy original seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_original_seedtable[j];
                }
                else
                    // copy current seedtable
                    for(j=0;j<Np;j++)
                        SEED_y0[j] = Internal.mtnic_current_seedtable[j];
            }
    }
}
```



## INITIALIZATION

- define events you want to profile
- reserve room for counter values
- initialize library
- program each event separately

```
#include "papi.h"

#define PCHECK(e) \
if (e!=PAPI_OK)
{printf("Problem in papi call, line %d\n",__LINE__); return 1;}

#define NEVENTS 3

int main(int argc, char **argv)
{
    int events[NEVENTS] =
    {
        PAPI_TOT_CYC,/* total cycles */
        PAPI_L1_DCM, /* stalls on L1 cache miss */
        PAPI_L2_DCM, /* stalls on L2 cache miss */
    };

    long_long int values[NEVENTS];
    int retval;

    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT)
        printf("wrong PAPI initialization: %d instead of %d\n", retval, PAPI_VER_CURRENT);

    {
        for ( int i = 0; i < NEVENTS; i++ )
        {
            retval = PAPI_query_event(events[i]) ;
            PCHECK(retval);
        }
    }
}
```



## (very) Basic usage of PAPI

```
retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
/* run the experiment */
for (i=0; i<NRUNS; i++)
{
    for (j=0; j<size; j++) array[j] = 2.3*array[j]+1.2;
}

retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);

printf("size: %d cycles: %lld cycles_per_loc: %9.5f L1miss: %lld <..>
       size,
       values[0], values[0]/(1.*NRUNS*size),
       values[1], (double)values[1]/(NRUNS*size),
       values[2], (double)values[2]/(NRUNS*size));
```

Start, stop and access

# Outline



## Addendum II

### Using gperftools

```
if(q > 0)
{
    switch(q)
    {
        case(1): // row y = 0 and/or plane symmetry
            if(subregions[i][BOTTOM][y] == 0)
                // this subregion in quadrant 1 contains the
                // set-up corners for seed sub-region gen
                SBL[y][0] = 0, SBL[X_0] = Nmesh - subregion
                STR[y][1] = 1, STR[X_0] = Nmesh - subregion
                // find horizontal extension in this quad
                Np = STR[X_0] - SBL[X_0];
                // allocate memory for seeds in this strip
                SEED_y0 = (unsigned int*)malloc(sizeof(unsigned
                int)*Np);
                if(!Internal.mtnic_original_seedtable)
                {
```



gperftools

# Use gperftools to obtain a basic profile

## OPTION I

- Include gperftools/profiler.h:

```
#include <gperftools/profiler.h>
```

- Encompass the code segments to be profiled within calls:

```
ProfilerStart ( "name_of_profile_file" );
...
ProfilerStop ( );
```

- Then variable CPUPROFILEFREQUENCY=x modifies the sampling frequency (x is in Hz)

## OPTION II

- link exec. against libprofiler.so

```
cc source.c -o exec -lprofiler
```

- or pre-load the profiler library

```
LD_PRELOAD=/path/to/libprofiler.so ./exec
```

- set variables CPUPROFILE to start the profiling

```
LD_PRELOAD=/path/to/libprofiler.so \
CPUPROFILE=./exec.prof CPUFREQUENCY=1000 \
./exec parameters
```



gperftools

# Use gperftools to obtain a basic profile

## READ THE GPERFTOOLS DATA

Use pprof, interactive mode:

```
pprof ./exec ./exec.prof
```

... let's have a live demonstration about:

top

- list
- disasm
- weblist
- gv / web

You can produce output on stdio:

```
pprof --text ./exec ./exec.prof  
pprof --text --lines ./exec ./exec.prof  
pprof --text --functions ./exec ./exec.prof
```

or in other image format

```
pprof <--gv|--web|--dot|--ps|--svg|--gif>
```

or output to be pipelined elsewhere

```
pprof <--raw|--collapsed|--callgrind>
```

or the whole call stack in detail

```
pprof --text --stack
```