

Linux Systems and Open Source Software

Basics of Performance Analysis Part II

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2022

Courtesy of:
Linux tracing systems & how they fit together, 2017
Stop the Guessing: Performance Methodologies for Production Systems, 2013
The TSA Method, 2014
Performance Analysis Methodology, 2018



Outline

- Overview of Performance Tools
- Sampling-based Profiling
- Instrumentation-based Profiling
- Analyses Performance w/ Different Data Sources
  - Dynamic Tracing
  - Tracepoint
  - Hardware Events



Overview

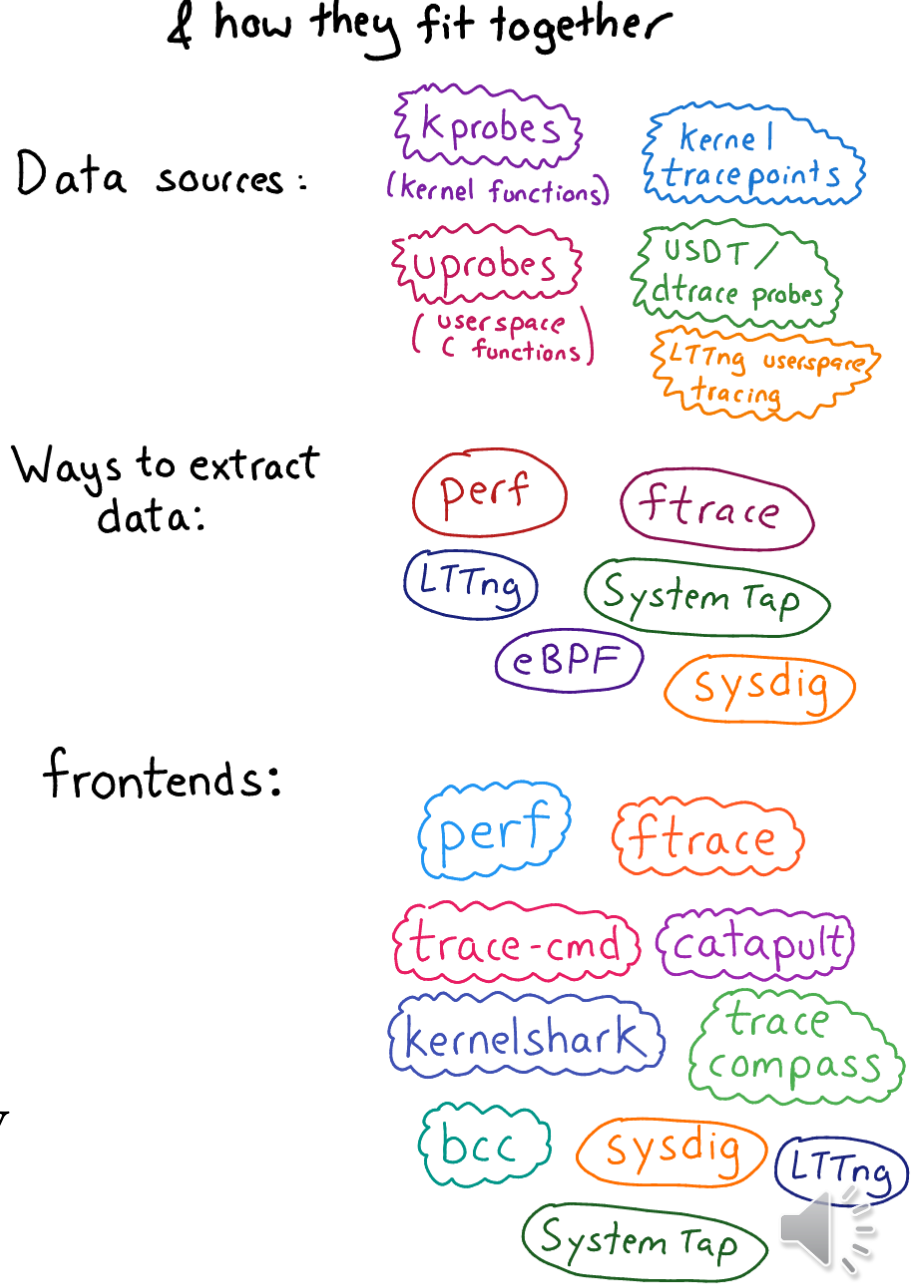
- There are plenty of performance tools available on Linux systems
  - strace, and ltrace, kprobes, and tracepoints, and uprobes, and ftrace, and perf, and eBPF

- We are going to talk about their
  - **internal mechanisms** (how they get performance data) and
  - **purposes** (i.e., functionalities)

A Closer Look of the Performance Tools

- It will be easy for us to understand the internals of the tools by splitting they into different parts

1. **Data sources:** Where the tracing data comes from
2. **Data collection (ways to extract data):** mechanisms for collecting data for those sources (like “ftrace”)
3. **Frontends:** the tool you actually interact with to *collect* and *analyze* data





Two Major Mechanisms to Get Performance Data

- Based on the **mechanism** the data are collected, the performance tools can be categorized into two classes:
  1. Sampling
  2. Instrumenting (or Tracing)
  - Some frontend tools supports both mechanisms to collect data

- There is usually a trade-off among
  - accuracy,
  - speed,
  - granularity of information, and
  - intrusiveness



SAMPLING-BASED PROFILING



Sampling-based Profiling

- **Sampling**
  - Monitoring CPU (or VM) **at regular time intervals** and **saving a snapshot of the HW/SW states**
    - E.g., contents of CPU registers, memory data, or software stack status
  - This data is then compared with the program's layout in memory to get an idea of where the program was at each sample
  - Time- and event-based sampling methods
- Pros:
  - No modification of SW is necessary
- Cons:
  - Less accurate
    - Varying sample interval leads to a **time/accuracy** trade-off
    - A high sample rate is accurate, but takes a lot of time for doing data collecting and data storaging
  - Very ***small* methods** will almost always be missed
    - if a small method is called frequently and you are unlucky, small but expensive methods may never show up

Time-based Sampling



- Data source: **Program Counter** (PC) in CPU

1. Usually, the performance tool registers a **call back function** to the Timer
   - The call back function can be seen as the Probes of the tracing tools
2. The Timer function will periodically trigger the execution of the **call back function**
   - The time interval is often set to 0.01 seconds
3. The call back function records the current value in PC
   - The PC values will be converted into the **function names** based on the debug information given in the program binaries
   - The ratio of the exe. time of each function is calculated by dividing the number of occurrence of each function against the total samples

- **gprof** is a common tool along with the GCC toolchain
- You compile your source program with the flags "**-pg**"
- The tool will sample your program during its execution
- gprof provides a flat profile to show the time ratios for **every sampled functions**

```
$gprof -a postmaster data/gprof/38202/gmon.out >result.out
$
$more result.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    33020     0.00     0.00  LWLockInitialize
  0.00      0.00     0.00     6721     0.00     0.00  SHMQueueInsertBefore
  0.00      0.00     0.00     5280     0.00     0.00  RestoreTransactionSnapshot
  0.00      0.00     0.00     4979     0.00     0.00  pg_wchar_strlen
  0.00      0.00     0.00     1868     0.00     0.00  SHMQueueInit
  0.00      0.00     0.00     1381     0.00     0.00  GUC_yylex
  0.00      0.00     0.00     1002     0.00     0.00  AtProcExit_LocalBuffers
  0.00      0.00     0.00      782     0.00     0.00  load_tzoffsets
  0.00      0.00     0.00      709     0.00     0.00  pg_tolower
  0.00      0.00     0.00      568     0.00     0.00  BufferShmemSize
  0.00      0.00     0.00      567     0.00     0.00  LockBufHdr
  0.00      0.00     0.00      567     0.00     0.00  ReadBufferWithoutRelcache
  0.00      0.00     0.00      567     0.00     0.00  finish_spin_delay
  0.00      0.00     0.00      504     0.00     0.00  ShmemAlloc
```

Courtesy of https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781784392970/16/ch16lvl1sec109/profiling-using-gprof



Event-based Sampling w/ Hardware Performance Counters

- Event-based sampling works similarly with the time-based sampling
  - Except that the **data sources are the hardware/software events & the PC values**

- These HW events are provided by hardware performance counters
  - **Also referred to as Performance Monitoring Units (PMU)**
  - The registers in PMU of the Intel Xscale processor are listed in the table below
    - Refer to page 152, 3rd Generation Intel Xscale Microarchitecture Developer's Manual, May 2007
    - Refer to Hardware Event-based Sampling Collection, Intel Vtune Amplifier User Guide

Table 84. Performance Monitoring Registers

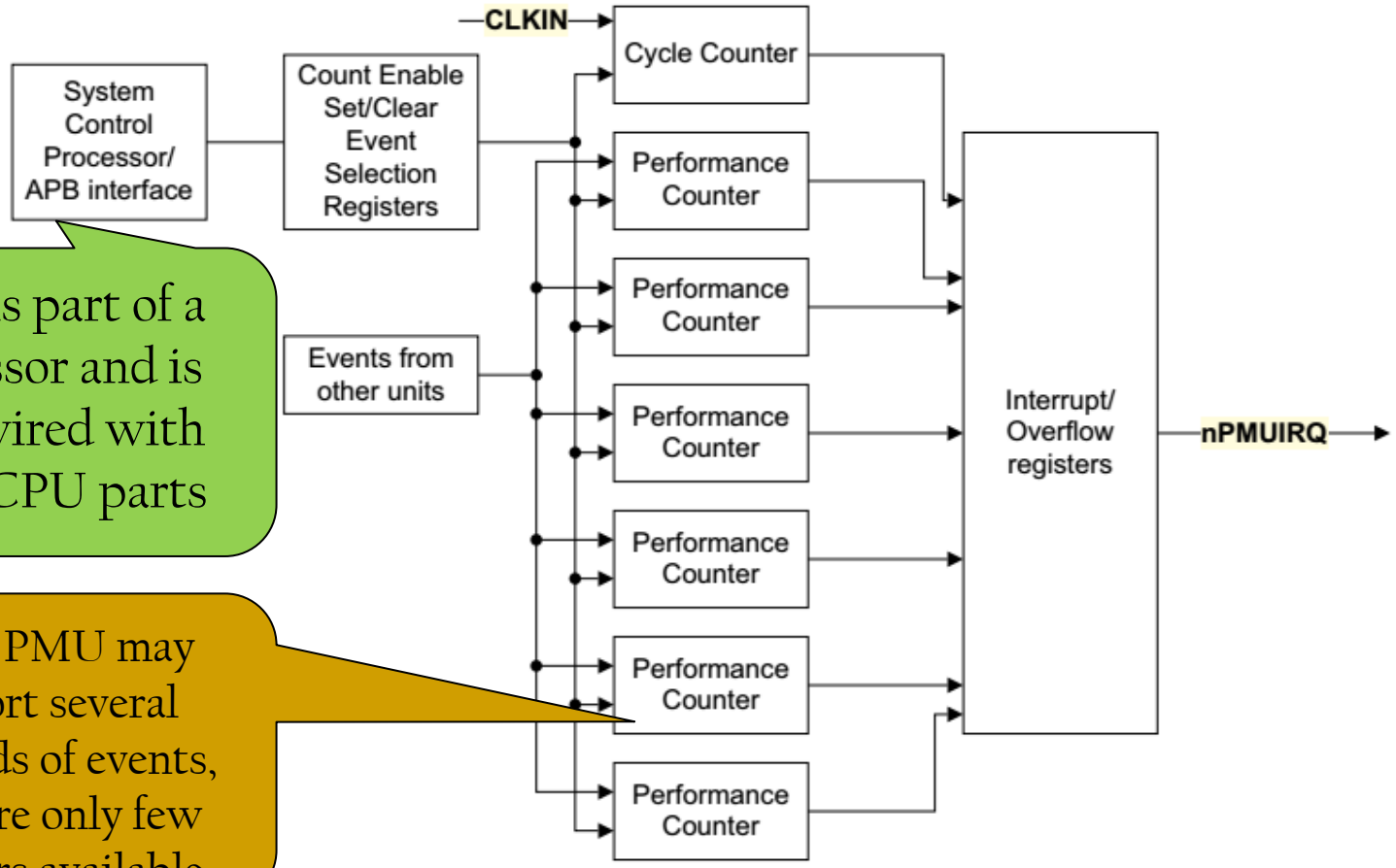
CRn	CRm	Access	Description	Cross-Reference
0	1	Read / Write	Performance Monitor Control Register	Section 11.2.1, page 11.0-154
1	1	Read / Write	Clock Counter Register	Section 11.2.2, page 11.0-155
4	1	Read / Write	Interrupt Enable Register	Section 11.2.3, page 11.0-156
5	1	Read / Write	Overflow Flag Register	Section 11.2.4, page 11.0-157
8	1	Read / Write	Event Selection Register	Section 11.2.5, page 11.0-158
0	2	Read / Write	Performance Count Register 0	Section 11.2.6, page 11.0-159
1	2	Read / Write	Performance Count Register 1	
2	2	Read / Write	Performance Count Register 2	
3	2	Read / Write	Performance Count Register 3	



Hardware Performance Counters (Cont'd)

- Modern processors built with performance counters for observing HW/SW interactions
  - E.g., total instruction count and mix, branch events, load/store events, L1/L2 cache events, prefetching events, TLB events, Multicore events
- Intel processors support at least 100's of types of events
- In isolation, events may not tell you much
- Event ratios are dynamically calculated values based on events that make up the formula
  - Cycles per instruction (CPI) consists of clockticks and instructions retired
  - There are a wide variety of predefined event ratios

Schematic of Performance Monitoring Unit



PMU is part of a processor and is hard wired with other CPU parts

While PMU may support several hundreds of events, there are only few counters available

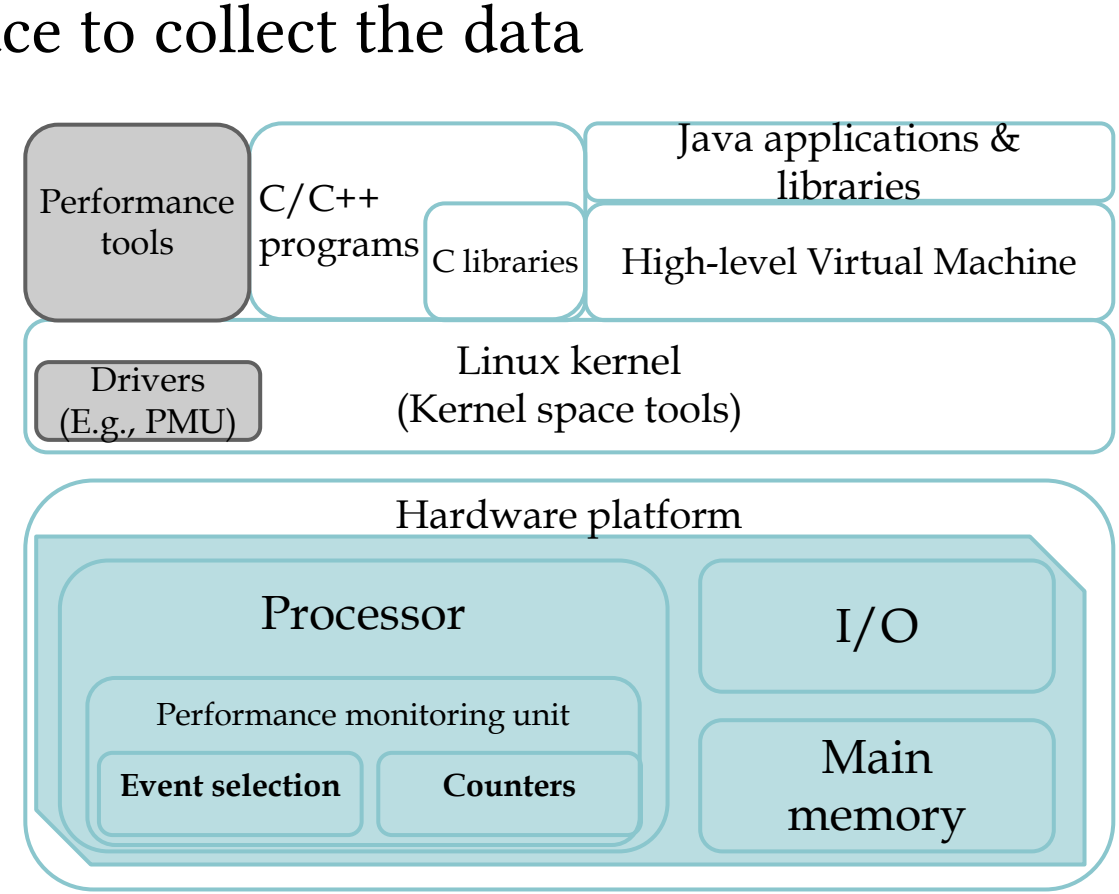
Figure 12-1 PMU block diagram

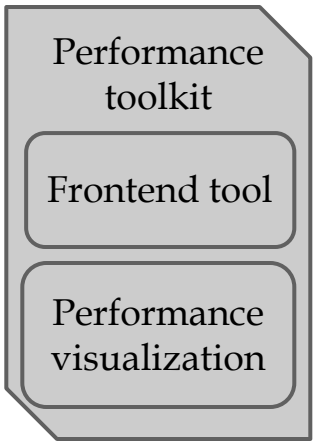
Courtesy of http://www.cnblogs.com/arnoldlu/p/7406882.html



Components of Perf. Tools

- Usually the performance tools are running on user-space
- It is common that some tools rely on the drivers in kernel-space to collect the data







INSTRUMENTATION-BASED PROFILING



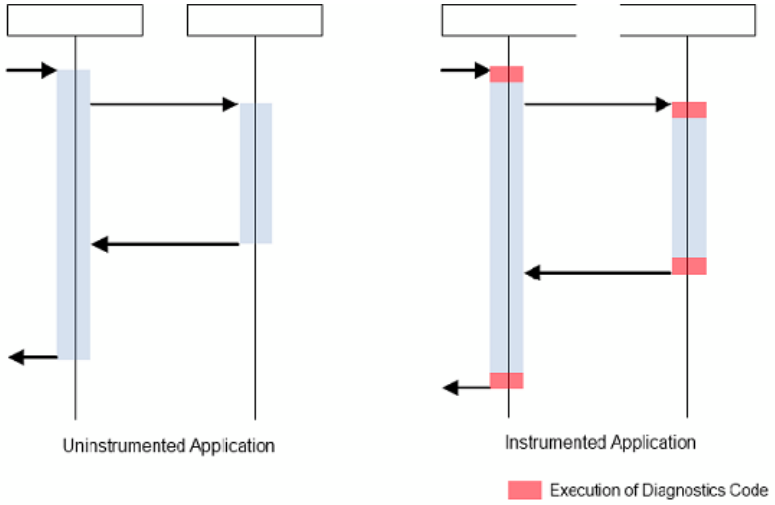
Instrumentation-based Profiling

- **Instrumentation**
  - Place special *profiling measurement code* into your SW manually or automatically offline/online
  - Is able to track the activities with *short burst* time
  - **This file focuses on the sampling-based method**
  - More information about the instrumentation-based method is given in the *Basics of Performance Analysis Part I*
- Pros:
  - Can be used across a variety of platforms
  - Very accurate (in some ways)
    - E.g., can capture the short-duration functions
- Cons:
  - Require recompilation or relinking of the SW
  - The inserted profiling code may affect performance
    - Difficult to calculate exact impact



Instrumentation-based Profiling (Cont'd)

- **Probe**
  - A debug statement(s) inserted into the target software (i.e., the red blocks shown in the figure) to help explore execution characteristics of software
  - i.e., the execution flow and state of software data structure at the moment a probe statement executes
  - **printk** is **the simplest form of probe statement** and one of the fundamental tools used by developers for kernel hacking
- **Instrumentation**
  - the actions of inserting the probes into the programs
  - The code insertion can be done offline (statically by human or compiler) or on-line (dynamically)
- **Trace**
  - (Verb.) the actions (or performance logging itself) listed above
  - (Noun) the collected data by tracing tools
  - We will use **Trace** in the following slides to refer to the action of logging program behaviors



Code example of the Probe

```
void probe () {
        t = time();
        printf("%d ", &t);
}
```



The Targets for Trace

- There are different targets for you to trace during system execution
  - **Userspace function calls**
    E.g., check if the allocated memory are freed by tracking **malloc** and **free** function calls
  - **System calls**
  - **Linux kernel function calls**
    E.g., which functions in my TCP stack are being called?
  - **Custom "events"** that you've defined either in userspace or in the kernel
  - **Instruction-level tracing**



Dynamic tracing

Tracepoint

Hardware events

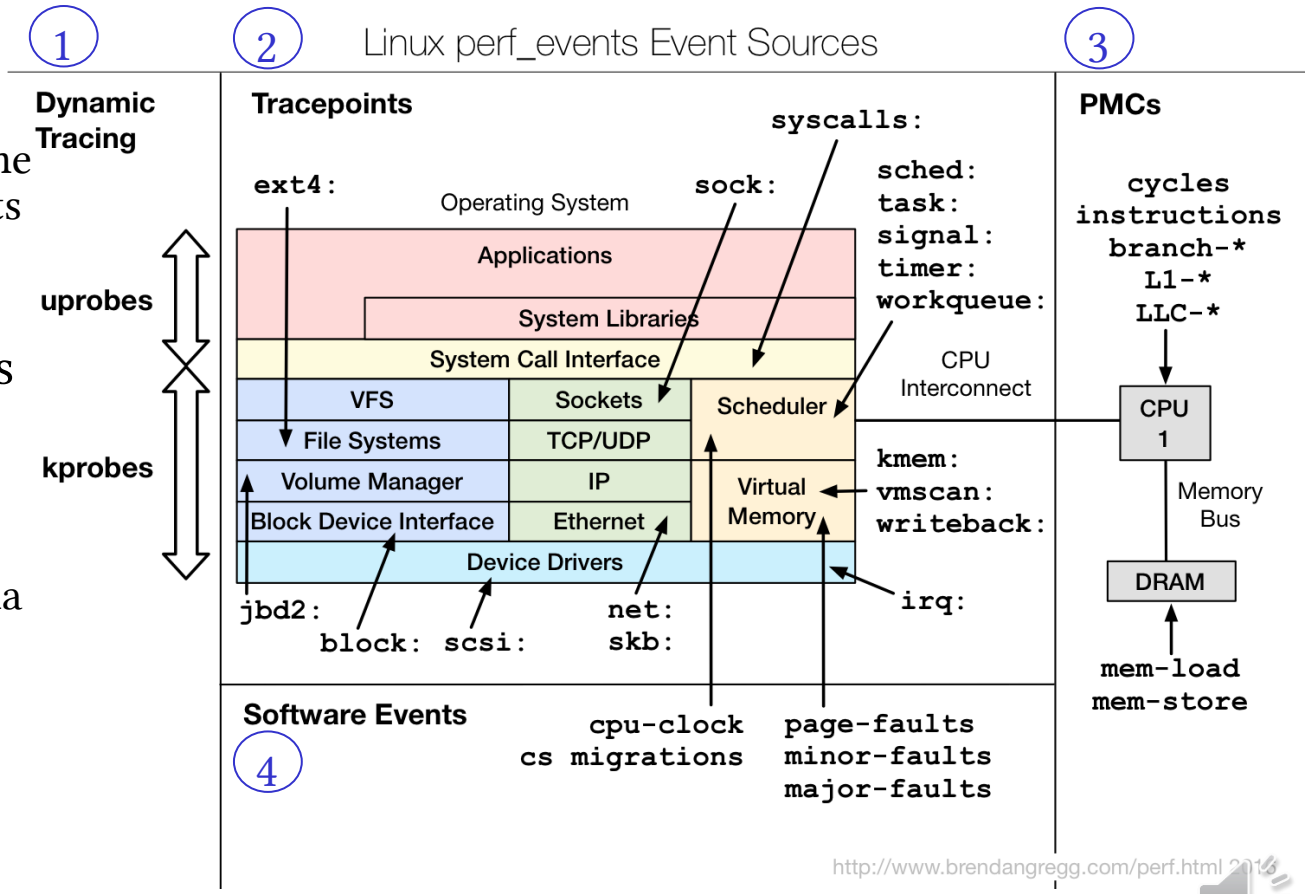
DATA SOURCES AND TOOLS FOR DATA COLLECTIONS & DISPLAY



Data Sources (Used by perf_events)

- Frontend tools are tied to specific data sources
  - Check the web link, perf Examples, for the usages of perf_events

- **perf_events** supports a wide diversity of events on Linux systems
  1. **Dynamic tracing** (via probes)
  2. **Tracepoints**
  3. **Software events**
  4. **PMCs** (Performance Monitoring Counters from the PMUs)





Data Sources (Cont'd)

1.

Dynamic Tracing: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the kprobes framework. For user-level software, uprobes.

2.

Kernel Tracepoint Events: This are ***static*** kernel-level instrumentation points that are hardcoded in interesting and logical places in the kernel.

User Statically-Defined Tracing (USDT): These are ***static*** tracepoints for user-level programs and applications.

3.

Software Events: These are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, etc.

4.

Hardware Events: CPU performance monitoring counters.

Timed Profiling: Snapshots can be collected at an arbitrary frequency, using **perf record -F** *Hz*

- This is commonly used for CPU usage profiling, and works by creating custom timed interrupt events

- The capabilities of perf_events are enormous, and you're likely to only ever use a fraction.

Courtesy of perf Examples, 2016



List of Events Supported by perf_events

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                       [Hardware event]
  instructions                               [Hardware event]
  cache-references                           [Hardware event]
  cache-misses                               [Hardware event]
  branch-instructions OR branches            [Hardware event]
  branch-misses                              [Hardware event]
  bus-cycles                                 [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  ref-cycles                                 [Hardware event]
  cpu-clock                                  [Software event]
  task-clock                                 [Software event]
  page-faults OR faults                      [Software event]
  context-switches OR cs                     [Software event]
  cpu-migrations OR migrations               [Software event]
  minor-faults                               [Software event]
  major-faults                               [Software event]
  alignment-faults                           [Software event]
  emulation-faults                           [Software event]
  L1-dcache-loads                            [Hardware cache event]
  L1-dcache-load-misses                      [Hardware cache event]
  L1-dcache-stores                           [Hardware cache event]
[...]
  rNNN                                       [Raw hardware event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier          [Raw hardware event descriptor]
   (see 'man perf-list' on how to encode it)
  mem:<addr>[:access]                        [Hardware breakpoint]
  probe:tcp_sendmsg                          [Tracepoint event]
[...]
  sched:sched_process_exec                   [Tracepoint event]
  sched:sched_process_fork                   [Tracepoint event]
  sched:sched_process_wait                   [Tracepoint event]
  sched:sched_wait_task                      [Tracepoint event]
  sched:sched_process_exit                   [Tracepoint event]
[...]
# perf list | wc -l
    657
```

Courtesy of perf Examples, 2016



Dynamic Tracing (Probes)

- A **probe** is when the kernel **dynamically modifies your assembly program at runtime** in order to enable tracing
  - by running the probe statements before/after the original instruction is executed

- You can enable a probe on literally any instruction in the program you're tracing
  - Kprobes and uprobes are examples of this pattern
  - Though *dtrace probes* are not "probes" in this sense



Data Source: uprobe

- We want to track the function calls to **func_1** and **funct_2** in the **test.c**
- Uprobe is used to insert the probes to the ***entries*** of these functions
  1. It is done by searching the program binary (**test.out** or **a.out**) for the **addresses** of these functions,
  2. Uprobe updates the code at the ***entrance addresses*** of these functions with the probe statements
     More about the dynamic trace (with Ftrace) is available here or there
  3. The probe statements are executed (and the corresponding data are dumped) whenever the ***entries*** of the functions are executed; the results can be displayed later

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015

Data Source: uprobe (Cont'd)

- Find the ***entries*** of these functions
  - We *compile* the program and *dump* the program binary to get the entrance **addresses** of the functions
    $\> gcc -o test.out test.c
    $\> objdump -d test.out
  - **0x400620** (entry of **func_1**) and **0x400640** (entry of **funct_2**)

```
0000000000400620 <func_1>:
  400620:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf6f8>
  400624:       912d4000        add     x0, x0, #0xb50
  400628:       b9400000        ldr     w0, [x0]
  40062c:       11000401        add     w1, w0, #0x1
  400630:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf6f8>
  400634:       912d4000        add     x0, x0, #0xb50
  400638:       b9000001        str     w1, [x0]
  40063c:       d65f03c0        ret

0000000000400640 <func_2>:
  400640:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf6f8>
  400644:       912d5000        add     x0, x0, #0xb54
  400648:       b9400000        ldr     w0, [x0]
  40064c:       11000401        add     w1, w0, #0x1
  400650:       90000080        adrp    x0, 410000 <__FRAME_END__+0xf6f8>
  400654:       912d5000        add     x0, x0, #0xb54
  400658:       b9000001        str     w1, [x0]
  40065c:       d65f03c0        ret
```

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static int func_1_cnt;
static int func_2_cnt;

static void func_1(void)
{
    func_1_cnt++;
}

static void func_2(void)
{
    func_2_cnt++;
}

int main(int argc, void **argv)
{
    int number;

    while(1) {
        sleep(1);
        number = rand() % 10;
        if (number < 5)
            func_2();
        else
            func_1();
    }
}
```

Assume the object code is for the ARM64 platform

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Capture Data w/ uprobe

1. Insert the probes to the ***entries*** of the functions
2. Activate the tracing
3. Collect the data during program execution

- tracefs file system is used to control the uprobe
  - Actually, some other tools can also be controlled via the same file system interface; e.g., kprobe
- When tracefs is configured into the kernel, the directory **/sys/kernel/tracing** will be created
  - Before 4.1, all ftrace tracing control files were within the debugfs file system, which is typically located at **/sys/kernel/debug/tracing**

- p means it is **a simple probe**
  - A probe can be either **simple** or **return,** r
- func_n_entry is the name we see in the trace output
  - Name is an optional field; if it is not provided, we should expect a name like p_test_0x644
- test is the executable binary in which we want to insert the probe
  - If test is not in the current directory, we need to specify path_to_test/test
- 0x620 or 0x640 is the **instruction offset** from the start of the program
  - The start address of the program is at 0x400000
- Please note >> in the second echo statement, as we want to add one more probe
- It will enable all the uprobe events when we write to **events/uprobes/enable**
  - We can enable individual events, as well, by writing to the specific event file created in the **events** directory

```
# 1. Insert probes to the function entries for the test program
$\> echo 'p:func_2_entry test:0x620' > /sys/kernel/debug/tracing/uprobe_events
$\> echo 'p:func_1_entry test:0x644' >> /sys/kernel/debug/tracing/uprobe_events

# 2. Activate the tracing facility to collect the data when the program is run
$\> echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable

# 3. Run the program
$\> ./test&
```

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015
tracefs file system for Ftrace, 2017



Display Uprobe Data

- The results are logged in the file: **/sys/kernel/debug/tracing/trace**
  - We can see what task was done by what CPU and at what time it executed the probed instruction
  - A function entry is printed out every second
  - This matches the code in the main function
- You can use similar commands to trace the ***returns*** of the functions

```
# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 8/8   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID CPU#  ||||  TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          test-2788  [003] ....  1740.674740: func_1_entry: (0x400644)
          test-2788  [003] ....  1741.674854: func_1_entry: (0x400644)
          test-2788  [003] ....  1742.674949: func_2_entry: (0x400620)
          test-2788  [003] ....  1743.675065: func_2_entry: (0x400620)
          test-2788  [003] ....  1744.675158: func_1_entry: (0x400644)
          test-2788  [003] ....  1745.675273: func_1_entry: (0x400644)
          test-2788  [003] ....  1746.675390: func_2_entry: (0x400620)
          test-2788  [003] ....  1747.675503: func_2_entry: (0x400620)
```

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Data Source: Perf
Inserting uprobe with Perf

- Finding the offset of an instruction or function for inserting a probe is cumbersome
- **Perf** is a useful tool to help prepare and insert uprobe into any line in source code
- Besides **perf**, there are a few other tools, such as **SystemTap**, **DTrace**, and **LTTng**, that can be used for kernel and user space tracing
  - however, **perf** is fully coupled with the kernel, so it's favored by kernel developers

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Capture Data w/ Perf (Cont'd)

Create probes (events)

- Use the sub-command, **probe**, offered by **perf**
  - Insert a probe point directly into the **function start** and **return**, a specific **line number of the source file**, etc.
  - You can get **a local variable** printed
  - You can have many other options, like all the instances of a function calling (see man perf probe for details)
- Add the created probes (events) and run the /test executable with **perf record**

Collect events and report data

- The command **perf record** is also responsible for collecting the data
  - When we create a perf probe point, we can have other recording options like **perf stat**, and
  - we can have many post-analysis options like **perf script** or **perf report**

```
# Build the program binary.
$\> gcc -g -o test test.c

# Create probes (events)
$\> perf probe -x ./test func_2_entry=func_2
$\> perf probe -x ./test func_2_exit=func_2%return
$\> perf probe -x ./test test_15=test.c:15
$\> perf probe -x ./test test_25=test.c:25 number

# Run the program and collect the data for the events.
$\> perf record -e probe_test:func_2_entry -e
probe_test:func_2_exit -e probe_test:test_15  -e
probe_test:test_25 ./test
```

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Display Perf Data

- The example below shows the data presented by **perf script**
- Similar data are presented here
  - When number < 5, func_2 is called
  - Otherwise, func_1 is called

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static int func_1_cnt;
static int func_2_cnt;

static void func_1(void) {
    func_1_cnt++;
}

static void func_2(void) {
    func_2_cnt++; // Line 15
}

int main(int argc, void **argv) {
    int number;
    while(1) {
        sleep(1);
        number = rand() % 10;
        // Line 25
        if (number < 5)     func_2();
        else                func_1();
    }
}
```

```
$\> perf script
        test  2741 [002]   427.584681: probe_test:test_25: (4006a0) number=3
        test  2741 [002]   427.584717: probe_test:test_15: (400640)
        test  2741 [002]   428.584861: probe_test:test_25: (4006a0) number=6
        test  2741 [002]   428.584872: probe_test:func_2_entry: (400620)
        test  2741 [002]   428.584881: probe_test:func_2_exit: (400620 <- 4006b8)
        test  2741 [002]   429.585012: probe_test:test_25: (4006a0) number=7
        test  2741 [002]   429.585021: probe_test:func_2_entry: (400620)
        test  2741 [002]   429.585025: probe_test:func_2_exit: (400620 <- 4006b8)
```

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Data Source: kprobe

- Similar to uprobe, to use kprobe we need to find **the symbols** (of kernel functions)
- Which are available in the file: **/proc/kallsyms**
  - other symbols that are not in the file have been blacklisted in the kernel
- A kprobe insertion into the **kprobe_events** file for the symbols that are not compatible with a kprobe insertion should result in a write error
  - A probe can be inserted at **some offset from the symbol base**, as well
  - Like uprobe, we can also trace the **return of a function** using kretprobe
  - The value of **a local variable** can also be printed in trace output
- You can also do the above with **perf**

```
# disable all events, just to insure that we see only kprobe output in trace.
$\> echo 0 > /sys/kernel/debug/tracing/events/enable

# disable kprobe events until probe points are inseted.
$\> echo 0 > /sys/kernel/debug/tracing/events/kprobes/enable

# clear out all the events from kprobe_events, to insure that we see output for only those for which we have enabled
$\> echo > /sys/kernel/debug/tracing/kprobe_events


# insert probe point at kfree
$\> echo "p kfree" >> /sys/kernel/debug/tracing/kprobe_events

# insert probe point at kfree+0x10 with name kfree_probe_10
$\> echo "p:kree_probe_10 kfree+0x10" >>
/sys/kernel/debug/tracing/kprobe_events

# insert probe point at kfree return
$\> echo "r:kfree_probe kfree" >>
/sys/kernel/debug/tracing/kprobe_events


# enable kprobe events until probe points are inseted.
$\> echo 1 > /sys/kernel/debug/tracing/events/kprobes/enable
```

Courtesy of Dynamic tracing in Linux user and kernel space, 2017
*Linux uprobe: User-Level Dynamic Tracing, 2015



Display Kprobe Data

- The corresponding results are kept in the **trace** file
- The Linux system is running in the background while you turn on the event tracing the logging starts
  - I.e., the **kfree return** is captured first followed by another **kfree** call
  - You may sometimes find the data points you are looking for, by yourselves or by some another tool

```
[root@pratyush ~]# more /sys/kernel/debug/tracing/trace
# tracer: nop
# entries-in-buffer/entries-written: 9037/9037   #P:8
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||   TIMESTAMP  FUNCTION
#              | |       |   ||||      |         |
          sshd-2189  [002]     dn..  1908.930731: kree_probe: (__audit_syscall_exit+0x194/0x218 <- kfree)
          sshd-2189  [002]     d...  1908.930744: p_kfree_0: (kfree+0x0/0x214)
          sshd-2189  [002]     d...  1908.930746: kree_probe_10: (kfree+0x10/0x214)
```



Tracepoint

- A **tracepoint** is something you compile into your program
  - When someone using your program wants to see when that **tracepoint is hit and extract data**, they can "**enable**" or "**activate**" the tracepoint to start using it
- A tracepoint in this sense usually **doesn't cause any extra overhead** when it's not activated
  - and is relatively low overhead when it is activated
  - Kernel tracepoints, USDT ("dtrace probes"), and lttng-ust are all examples of this pattern



Kernel Tracepoints

- The tracepoint ***events*** are hardcoded into the source code of the software offline
  - They are placed at logical and interesting locations, and presented (event name and arguments) as a stable API
  - Kernel tracepoints are to expose the custom events in Linux kernel
  - In practice, they are often done by placing macros, such as TRACE_EVENT for kernel tracepoints
- **TRACE_EVENT** macro defines tracepoint event
  - The example tracepoint is inserted into the **UDP packet handling module in Linux kernel** and captures the event of "UDP … queue failures"

```
TRACE_EVENT(udp_fail_queue_rcv_skb,
        TP_PROTO(int rc, struct sock *sk),
    TP_ARGS(rc, sk),
    TP_STRUCT__entry(
            __field(int, rc)
            __field(__u16, lport)
    ),
```

The event (callback) is added in the source code

The list of added events can be displayed. See USDT example in the next page.

Courtesy of Linux tracing systems & how they fit together, 2017
perf Examples, 2016

Userland Statically Defined Tracing (USDT)

- USDT works similar as kernel tracepoints
- Usually, it is supported by some of the popular software, such as Node.js (node) and Python
- The right figure shows the partial tracepoints built in the Node.js executable
  - These tracepoints are added at some important functions for the users to understand the runtime behaviors of Node.js

```
$ readelf -n node
Displaying notes found at file offset 0x00000254 with length 0x00000020:
  Owner                Data size        Description
  GNU                  0x00000010       NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 2.6.32

Displaying notes found at file offset 0x00e70994 with length 0x000003c4:
  Owner                Data size        Description
  stapsdt              0x0000003c       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__start
    Location: 0x0000000000dc14e4, Base: 0x0000000000112e064, Semaphore: 0x000000000147095c
    Arguments: 4@%esi 4@%edx 8@%rdi
  stapsdt              0x0000003b       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__done
    Location: 0x0000000000dc14f4, Base: 0x0000000000112e064, Semaphore: 0x000000000147095e
    Arguments: 4@%esi 4@%edx 8@%rdi
  stapsdt              0x00000067       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__server__response
    Location: 0x0000000000dc1894, Base: 0x0000000000112e064, Semaphore: 0x0000000001470956
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000061       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: net__stream__end
    Location: 0x0000000000dc1c44, Base: 0x0000000000112e064, Semaphore: 0x0000000001470952
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000068       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: net__server__connection
    Location: 0x0000000000dc1ff4, Base: 0x0000000000112e064, Semaphore: 0x0000000001470950
    Arguments: 8@%rax 8@-1144(%rbp) -4@-1148(%rbp) -4@-1152(%rbp)
  stapsdt              0x00000060       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__client__response
    Location: 0x0000000000dc23c5, Base: 0x0000000000112e064, Semaphore: 0x000000000147095a
    Arguments: 8@%rdx 8@-1144(%rbp) -4@%eax -4@-1152(%rbp)
  stapsdt              0x00000089       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__client__request
    Location: 0x0000000000dc285e, Base: 0x0000000000112e064, Semaphore: 0x0000000001470958
    Arguments: 8@%rax 8@%rdx 8@-2184(%rbp) -4@-2188(%rbp) 8@-2232(%rbp) 8@-2240(%rbp) -4@-2192(%rbp)
  stapsdt              0x00000089       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__server__request
    Location: 0x0000000000dc2e69, Base: 0x0000000000112e064, Semaphore: 0x0000000001470954
    Arguments: 8@%r14 8@%rax 8@-4344(%rbp) -4@-4348(%rbp) 8@-4304(%rbp) 8@-4312(%rbp) -4@-4352(%rbp)
```

Courtesy of perf Examples, 2016

USDT w/ perf_events



- Static user tracing is supported in later 4.x series Linux kernels
  - The following demonstrates Linux 4.10 (with an additional patchset), and tracing the Node.js USDT probes*
  - **http__server__request** event, which is invoked for each incoming HTTP request

```
$\> perf list | grep sdt_node
  sdt_node:gc__done                          [SDT event]
  sdt_node:gc__start                         [SDT event]
  sdt_node:http__client__request             [SDT event]
  sdt_node:http__client__response            [SDT event]
  sdt_node:http__server__request             [SDT event]
  sdt_node:http__server__response            [SDT event]
  sdt_node:net__server__connection           [SDT event]
  sdt_node:net__stream__end                  [SDT event]
$\> perf record -e sdt_node:http__server__request -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.446 MB perf.data (3 samples) ]
$\> perf script
        node  7646 [002]   361.012364: sdt_node:http__server__request: (dc2e69)
        node  7646 [002]   361.204718: sdt_node:http__server__request: (dc2e69)
        node  7646 [002]   361.363043: sdt_node:http__server__request: (dc2e69)
```

*Refer to Sec. 6.5 from the page to know the context of the above example.

Courtesy of perf Examples, 2016



Hardware Events (Event Profiling)

- In addition to sampling at time intervals, taking samples triggered by PMUs is another form of profiling
  - which can be used to shed some light on **the hardware activities during the software execution**, such as cache misses, memory stall cycles, and other low-level processor events
  - Example: Hardware events supported by **perf_events** are listed below
- Two ways to collect HW data: **per-application** or **system-wide**

```
$\> perf list | grep Hardware
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
cache-misses [Hardware event]
branch-misses [Hardware event]
bus-cycles [Hardware event]
L1-dcache-loads [Hardware cache event]
L1-dcache-load-misses [Hardware cache event]
L1-dcache-stores [Hardware cache event]
L1-dcache-store-misses [Hardware cache event] [...]
```

Courtesy of perf Examples, 2016



Summarize HW Events w/ perf_events for An Application

- The **perf stat** command summarizes HW events during the execution of the ***program***, **gzip**
  - You can use the flag "**-d**" to get more detailed results

```
$\> perf stat gzip file1

 Performance counter stats for 'gzip file1':

      1920.159821 task-clock                #    0.991 CPUs utilized
               13 context-switches          #    0.007 K/sec
                0 CPU-migrations            #    0.000 K/sec
              258 page-faults               #    0.134 K/sec
    5,649,595,479 cycles                    #    2.942 GHz                      [83.43%]
    1,808,339,931 stalled-cycles-frontend   #   32.01% frontend cycles idle     [83.54%]
    1,171,884,577 stalled-cycles-backend    #   20.74% backend  cycles idle     [66.77%]
    8,625,207,199 instructions              #    1.53  insns per cycle
                                            #    0.21  stalled cycles per insn [83.51%]
    1,488,797,176 branches                  #  775.351 M/sec                    [82.58%]
       53,395,139 branch-misses             #    3.59% of all branches          [83.78%]

      1.936842598 seconds time elapsed
```

Courtesy of CPU Statistics in perf Examples, 2016



Select HW Events w/ perf_events for An Application

- Profile w/ the selected HW events, instead of the default set
  - The percentage printed is a convenient calculation that **perf_events** has included, based on the counters
  - I.e., dcache miss rate = **L1-dcache-misses/L1-dcache-loads**
  - If you include the "cycles" and "instructions" events, **perf stat** will report an IPC calculation in the output
- The hardware events that can be measured are often specific to the processor used to run the command
  - Many may not be available from within a virtualized environment

```
$\> perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores gzip file1

 Performance counter stats for 'gzip file1':

     1,947,551,657 L1-dcache-loads

       153,829,652 L1-dcache-misses #    7.90% of all L1-dcache Load misses
     1,171,475,286 L1-dcache-stores

      1.538038091 seconds time elapsed
```

Courtesy of perf Examples, 2016



Select HW Events w/ perf_events for Entire System

- The **perf record** command with the flag "**-e**" to specify the event(s) you want to observe (at ***function level***)
  - It can be HW or SW event
  - The profiling operation is terminated by **Ctrl-C** or Use **sleep** command to run the profiling for a specific time duration
  - Display data with **perf script** or **perf report**
- You can do the following profile on your own
  - Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds (specified as argument to **sleep**)
  - **$\> perf record -e LLC-load-misses -c 100 -ag -- sleep 5**
  - You can ***run*** the above command by yourselves and interpret the performance results
  - You can refer to this page to know the difference between **stat** and **record**



QUESTIONS



References

- Linux Tracing Technologies
- Linux uprobe: User-Level Dynamic Tracing
- Linux tracing systems & how they fit together
- perf examples
- Cheat sheet for perf
- Systems Performance Enterprise and the Cloud, 2nd Edition