

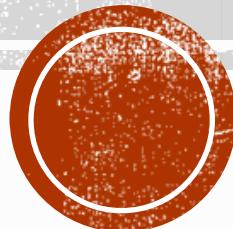
# Using Linux Trace Tools

For diagnosis, analysis, learning and fun...

**OLIVER YANG**

**JUL, 23 2016**

**[HTTP://OLIVERYANG.NET](http://OLIVERYANG.NET)**



# AGENDA

- Overview
- Trace Building Blocks
- Diagnosis, Analysis and Learning
- For Fun...



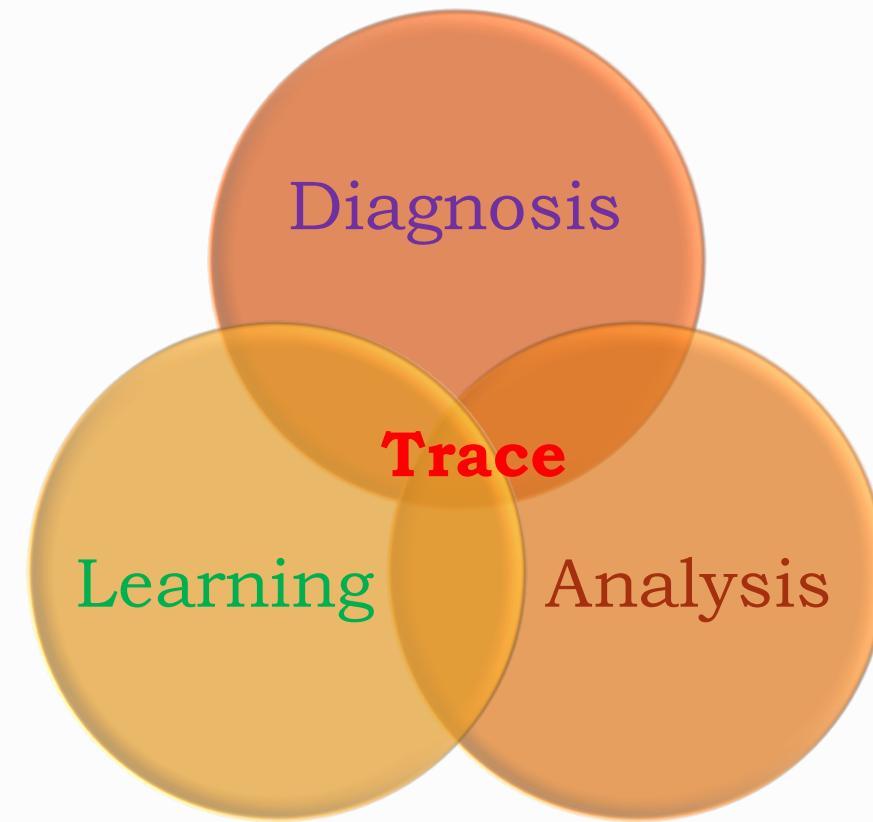
# What Is Dynamic Trace?

- Trace probe could be pre-defined or defined on the fly
- All probes are not enabled by default, zero performance impacts
- Users could enable one or multiple probes and hooks dynamically
  - Hook functions could be predefined or defined dynamically
  - Could be defined by CLI or programming scripts
- While probes get hit, hook functions could be invoked
  - Hook function could be following functions
    - Print trace logs
      - Event logs, stack trace, function graph, timestamp, duration calculation, histogram
    - Drop into kernel debugger (Not support by Linux?)
    - Panic to save core files



# Why Dynamic Trace?

- Classic use cases...
  - Functional issues diagnosis
  - Performance analysis
  - System behaviors learning
- New use cases...
  - NFV ([IOVISOR](#))
  - Security...



# Dynamic Trace - Pros vs. Cons

- Pros
  - Good coverage by massive probes
  - Available in production mode
  - Fine granularity, could be API or source code lines level
  - Optimized for less overhead
  - Building blocks for new tools development
  - Have community and ecosystem support
    - Trace toolkits and application support
- Cons
  - Tools supporting programming need significant learning time
    - Master a new script language
    - Skills to process massive tracing data
  - Some dynamic trace scripts could crash kernel, like SystemTap
  - Performance could be impacted if the trace scripts are not well designed



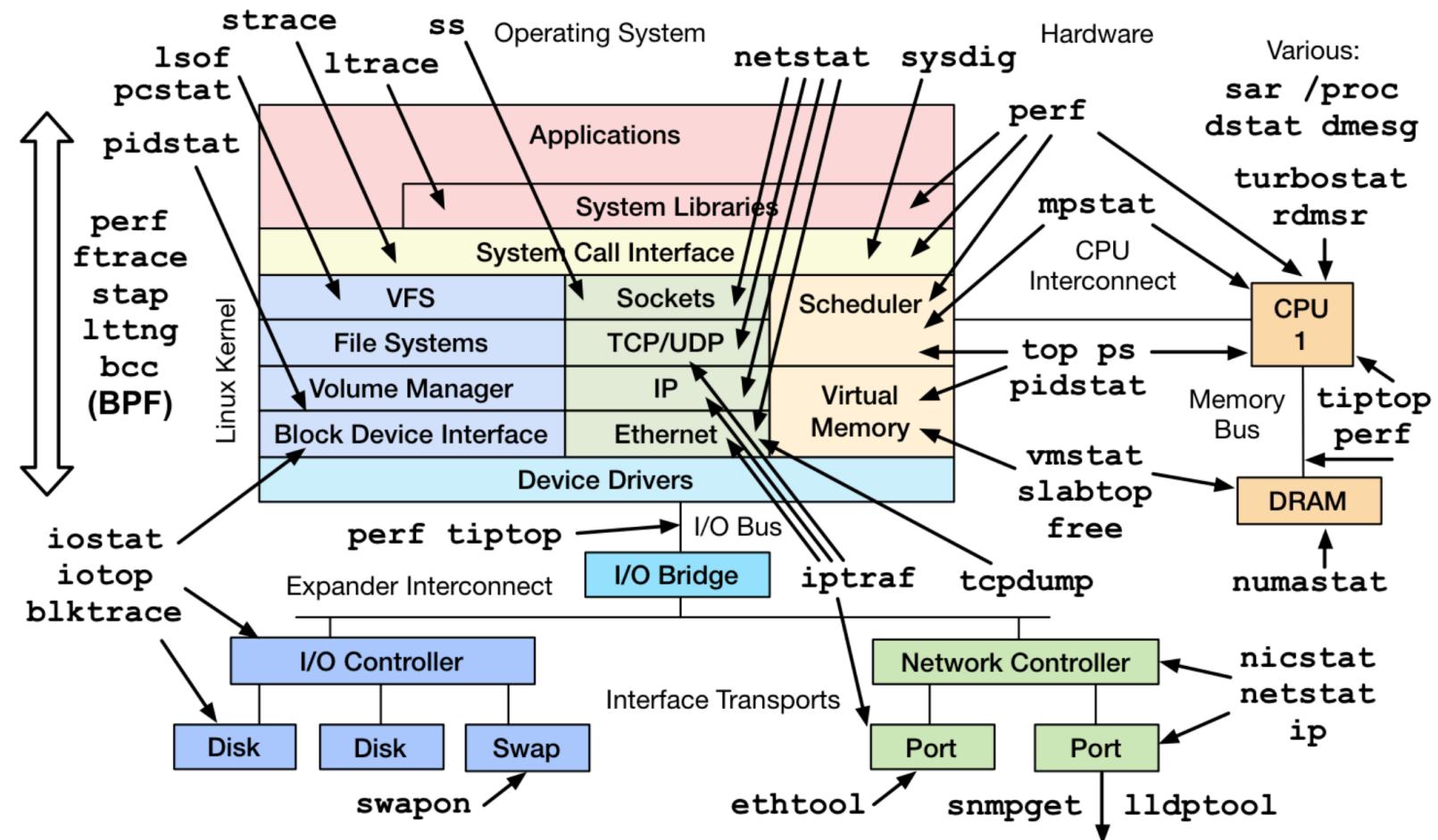
# Dynamic Trace history

- 2004: Linux kprobes (2.6.9)
- 2005: Solaris Dtrace (10)
- 2008: Linux ftrace (2.6.27)
- 2009: Linux perf (2.6.31)
- 2009: Linux tracepoints (2.6.32)
- 2012: Dtrace: Oracle Unbreakable Enterprise Linux (2.6.39)
- 2012: Linux uprobes (3.5)
- 2010-2016: ftrace, kprobe, perf\_events enhancements (3.7, 3.9 ...4.6)
  - Ftrace: mcount hook supports saving arguments
  - Kprobe: optimization by using ftrace mcount hook
- 2014-2016: eBPF patches (3.15,3.19,4.0,4.1...4.6)



# Big picture

Linux Performance Observability Tools

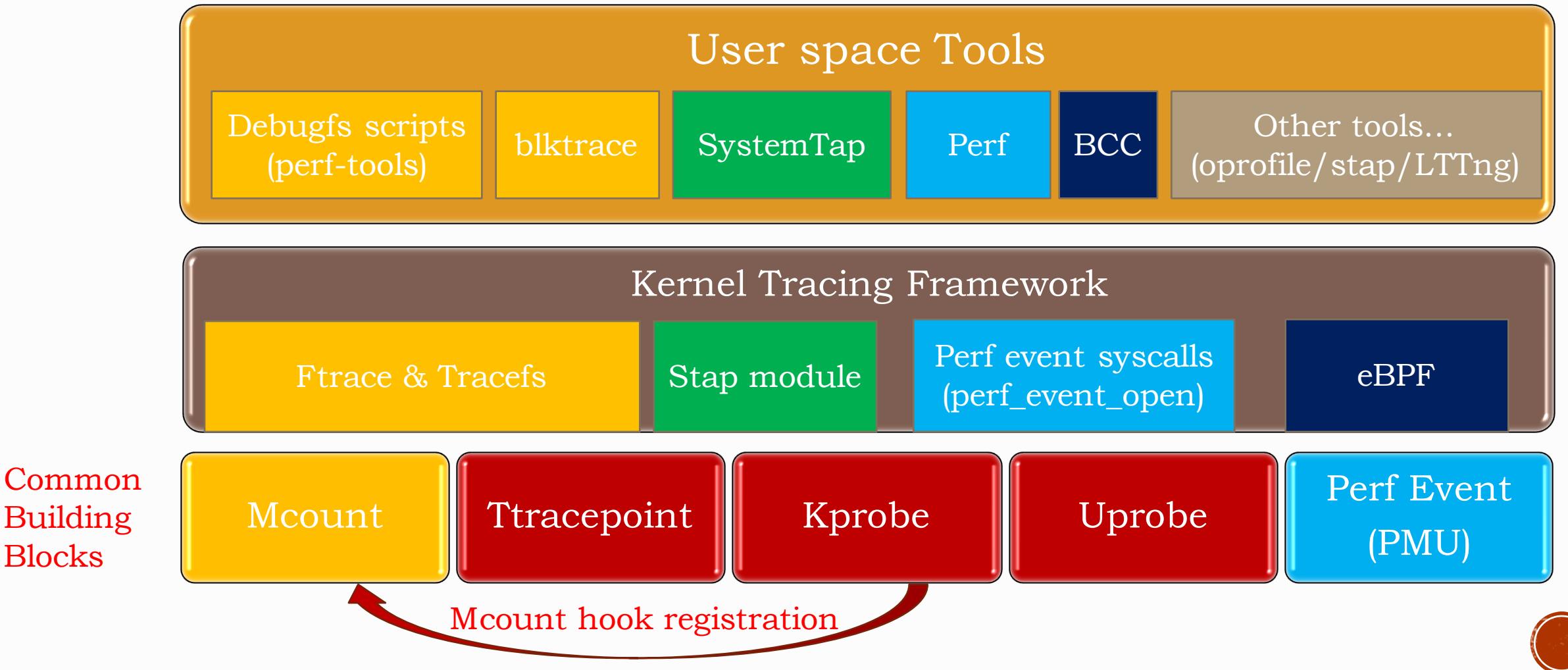


# AGENDA

- Overview
- Trace Building Blocks
- Diagnosis, Analysis and Learning
- For Fun...



# Conceptual View



# Predefined Trace Events

- Mcount
  - Kernel function entry trace
  - Implicit instrumentation at function entry point
    - Compiler insert instrumentations at kernel build time
      - By leveraging gcc -pg option
    - Replace instrumentation with noop instructions at kernel boot time
      - While tracer is enabled, noop got replaced by calling mcount hook function
  - Transparent to its users, not declarations in code
- Tracepoint
  - Kernel predefined trace probe
  - Explicit instrumentation in kernel code
    - Predefined by kernel & driver code, which need explicit declarations
    - By default off, but leaves the noop instructions for dynamic enabling
    - After enable, the noop instruction gets replaced by jump, which goes to real tracing code



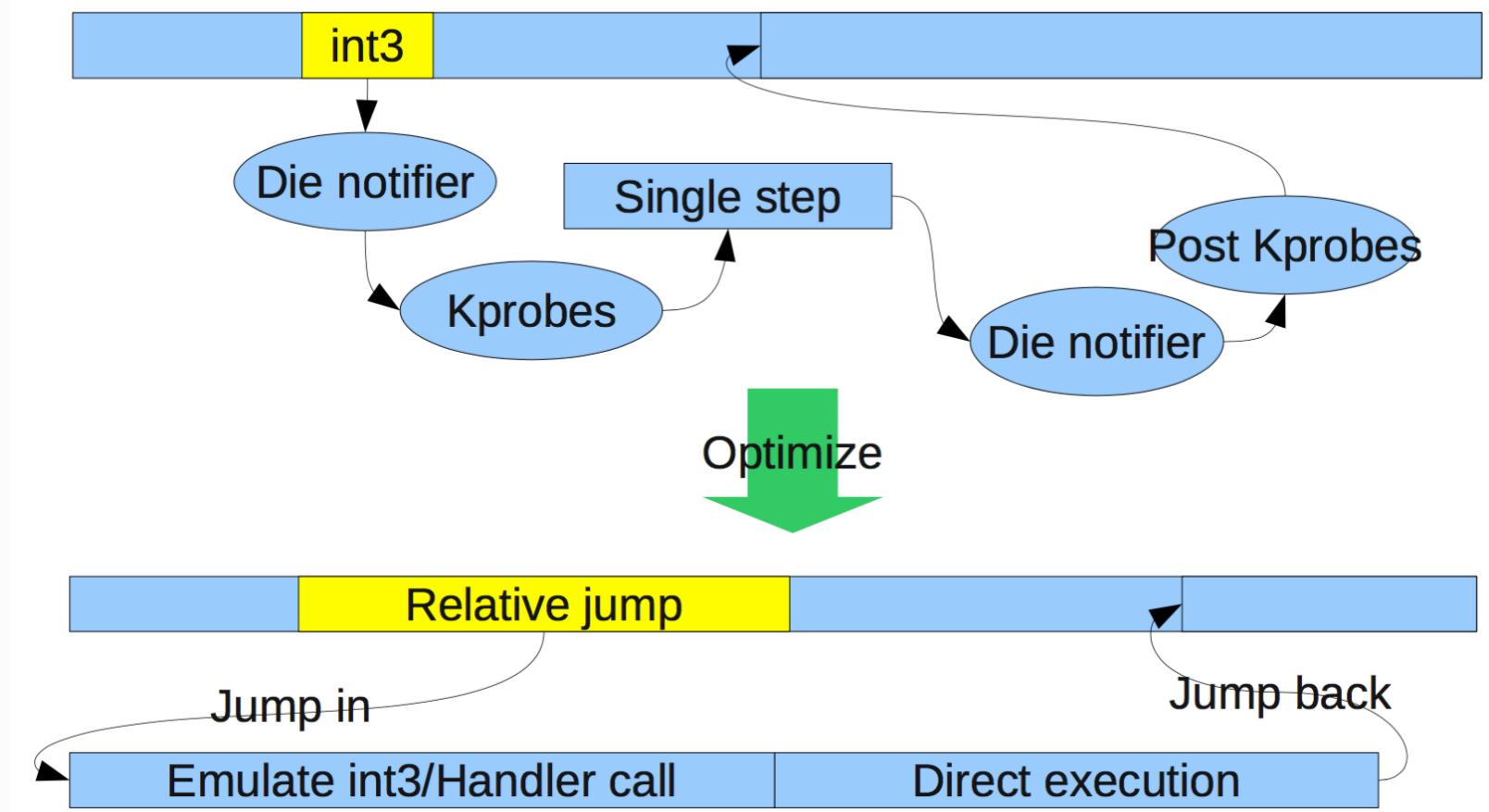
# Dynamic Software Trace Events

- Kprobe
  - Define a trace event in kernel dynamically
    - Replace the probed instruction with a breakpoint, mcount hook, or jump instruction
    - Hit the probe then jump to kprobe instruments
  - Three type of probes, for writing kernel modules based on C language
    - Kprobe: any instructions in a kernel function
    - Jprobe: entry point of a kernel function, for handling arg list (support mcount hook)
    - Return Probe: return point of a kernel function, for handling return address
- Uprobe
  - Define a trace event in user space dynamically
    - A user space counterpart to kprobe
    - A kernel hook can be invoked whenever a process executes a specific instruction location
  - Two type of probes
    - Uprobe: setting breakpoints for any instructions in user space functions
    - Uretprobe: hit the probe while return from user space function



# Trace Internal - Kprobe Example

- 3 core technologies
  - Binary hot patch
  - Breakpoint
  - Language support



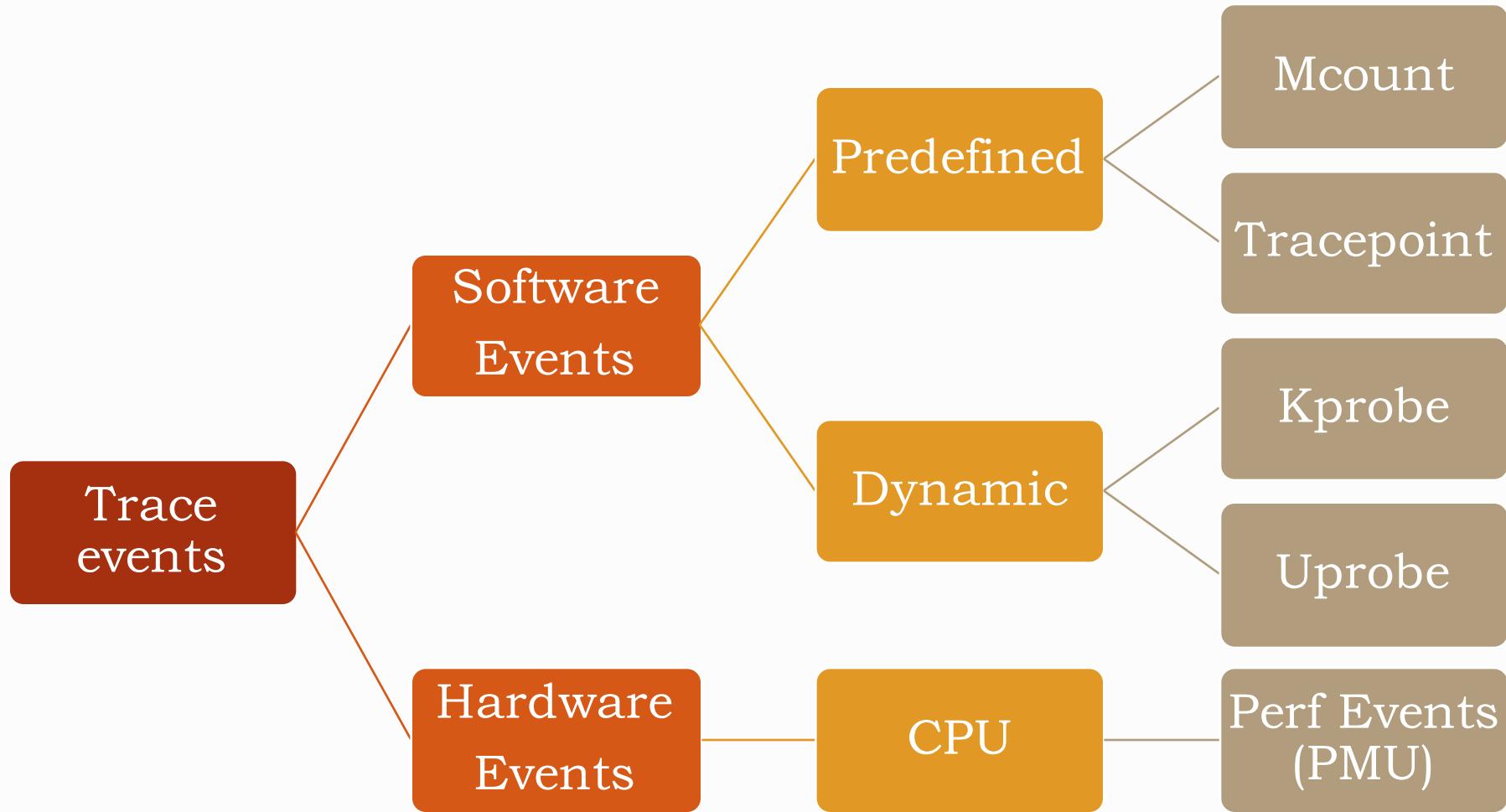
# Hardware Events

- Based on hardware mechanisms
  - CPU PMU registers
  - Hardware breakpoints
- Still works for VM
  - Depends on PMU virtualization
- Perf events
  - Unified syntax to refer the events
  - sudo perf list | grep -i hardware

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]
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]
L1-dcache-prefetches	[Hardware cache event]
L1-dcache-prefetch-misses	[Hardware cache event]
L1-icache-loads	[Hardware cache event]
L1-icache-load-misses	[Hardware cache event]
L1-icache-prefetches	[Hardware cache event]
L1-icache-prefetch-misses	[Hardware cache event]
LLC-loads	[Hardware cache event]
LLC-load-misses	[Hardware cache event]
LLC-stores	[Hardware cache event]
LLC-store-misses	[Hardware cache event]
LLC-prefetches	[Hardware cache event]
LLC-prefetch-misses	[Hardware cache event]
dTLB-loads	[Hardware cache event]
dTLB-load-misses	[Hardware cache event]
dTLB-stores	[Hardware cache event]
dTLB-store-misses	[Hardware cache event]
dTLB-prefetches	[Hardware cache event]
dTLB-prefetch-misses	[Hardware cache event]
iTLB-loads	[Hardware cache event]
iTLB-load-misses	[Hardware cache event]
branch-loads	[Hardware cache event]
branch-load-misses	[Hardware cache event]
node-loads	[Hardware cache event]
node-load-misses	[Hardware cache event]
node-stores	[Hardware cache event]
node-store-misses	[Hardware cache event]
node-prefetches	[Hardware cache event]
node-prefetch-misses	[Hardware cache event]
rNNN	[Raw hardware event descriptor]
cpu/t1=v1[,t2=v2,t3 ...]/modifier	[Raw hardware event descriptor]
mem:<addr>[:access]	[Hardware breakpoint]



# Summary - Trace Event Type



# AGENDA

- Overview
- Trace Building Blocks
- *Diagnosis, Analysis and Learning*
- For Fun...



# Functional Issues Diagnosis

- Troubleshooting
- Error monitoring
- Bug report and fixes
- Trace data presentation
  - Stack backtrace
    - Inputs arg values
    - Return value
    - Error handling branch
    - Print any global or local variables
  - Trace events related to errors



# Diagnosis Process

- A. Find a problem
- B. Make an assumption or ask a question
- C. Find the proper tracepoints or understand related source code
- D. Search a trace tool or write your own trace tool
  - May need consult from programming manuals
- E. Verify assumption by running tracing tools
- F. Trace results analysis
  - May need write a tool to process trace data
- G. Repeat step B until address the issues



# Diagnosis Example - 1

- A. rmmod: ERROR: Module sampleblk is in use
- B. Why did rmmod return the error? What was the error?
- C. Run strace rmmod to study rmmod process
  - Results showed that it is module reference code issue
  - module:module\_get trace point could help on this issue
- D. The tool tpoint could trace the trace points: module:module\_get
- E. Run tpoint and reproduce the bug

```
$ sudo ./tpoint module:module_get
```

Tracing module:module\_get. Ctrl-C to end.

```
systemd-udevd-2986 [000] .... 196.382796: module_get: sampleblk call_site=get_disk refcnt=2
```

```
systemd-udevd-2986 [000] .... 196.383071: module_get: sampleblk call_site=get_disk refcnt=3
```

- F. Got the tpoint logs, found that who is using the module
- Please find the detailed information from [Linux Block Driver – 1](#) section 4.2.



# Diagnosis Example - 2

- Problem: kprobe event format is difficult, any alternative solution?
- For example 2, using SystemTap could address the same issue,
- Query the do\_unlinkat available input arguments and local variables,

```
$ sudo stap -L 'kernel.function("do_unlinkat")'
```

```
kernel.function("do_unlinkat@fs/namei.c:3857") $dfd:int $pathname:char const* $path:struct path  
$last:struct qstr $type:int $delegated_inode:struct inode*
```

- Run below SystemTap one liner command,

```
$ sudo stap -e 'probe kernel.function("do_unlinkat") { printf("%s \n", kernel_string($pathname))}'
```

- Notes:

- **Green** is function name and source code information.
- **Red** is input arguments
- **Blue** is local variable
- Please find SystemTap book from [here](#).



# Diagnosis Example - 3

- Problem: An unauthorized application is removing SQLite journal file
- Find what could you probe, and list source code line by line,

```
# perf probe -L do_unlinkat | head -n2
<do_unlinkat@/lib/modules/4.6.0-rc3+/build/fs/namei.c:0>
    0 static long do_unlinkat(int dfd, const char __user *pathname)
# perf probe -V do_unlinkat | grep pathname
                char*      pathname
```

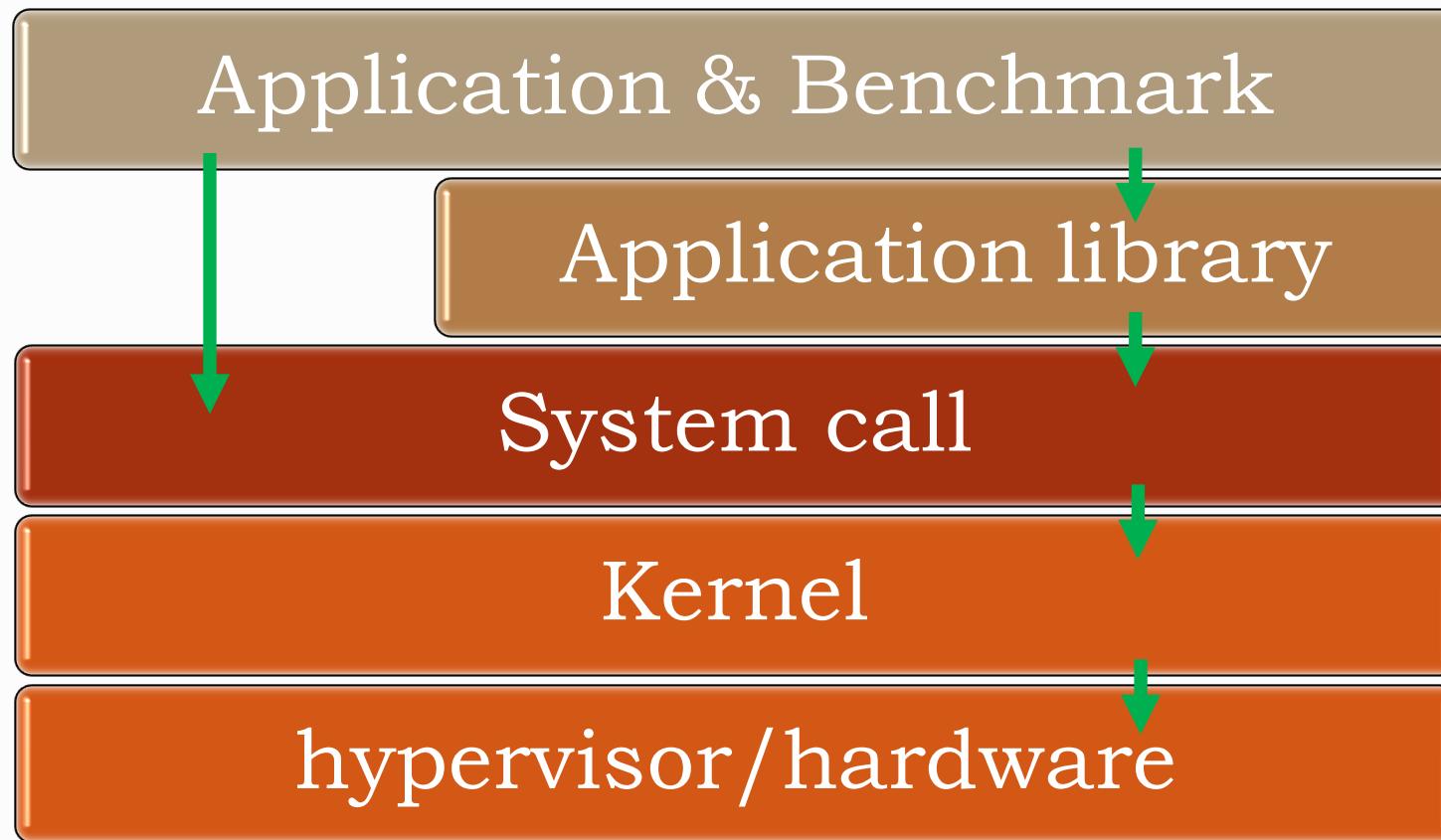
- Using perf probe to get the 2<sup>nd</sup> arg of do\_unlinkat

```
# perf probe --add 'do_unlinkat pathname:string'
```

- Monitor all file removing operation within 3600 seconds

```
# perf record -e probe:do_unlinkat -aR sleep 3600
# perf script
    sysmon 1964 [056] 237504.227636: do_unlinkat: (ffffffff810ed1b0)
    pathname="/ddr/registry/m0/alerts_floating.db3-journal"
    sysmon 1964 [056] 237509.233384: do_unlinkat: (ffffffff810ed1b0)
    pathname="/ddr/registry/m0/alerts_floating.db3-journal"
```

# Perf Analysis - Conceptual View



- App & Test Centric
- Resource & Machine Centric



# Perf Analysis - Methodologies

## App & Test Centric

- Problem statement method
- Functional diagram method
- Workload Characterization
- Workload Analysis
- RED Method

## Resource & Machine Centric

- Resource Analysis
- USE Method
- Kernel Resource Analysis



# Perf Analysis - Trace vs. Profiling

- Trace
  - Hit the probe, and record
  - Trace buffer could overflow, and trace events could be dropped
- Profiling
  - Sampling the probe events with certain frequency (typically 1HZ)
  - Driven by timer (1HZ = 1ms)
    - NMI interrupts – by CPU PMU NMI events, which is better
    - Timer interrupts
- Trace data presentation
  - Function graph
  - Timestamp or Latency calculations for each operations or functions
  - Stack back trace or function name statistic related to quick or fast path
  - Perf Histogram: latency, data size, trace events counters
  - Virtualized trace data
    - Excel (Line Chart, Histogram), Flamegraph, Heatmap



# Analysis - Workload Characterization

- **Who** should be perf analysis target?
  - Identify application (pid, tid, users)
- **Why** should it be analysis target?
  - Bad perf number or metrics?
    - Computing - Busy CPU
    - Storage - Throughput, IOPS
    - Network – RX/TX throughput, IP forwarding
- **How** to trace or profiling the target?
  - Reproduce problem by running app, micro-benchmark
  - Select tools or writing scripts
- **What** should be investigated for the target?
  - Computing - on/off CPU analysis, code path and context
  - Storage and Network - IO pattern analysis, code path, context



# Analysis - USE Method

- For every resource, check...
  - Utilization - busy time or sleep time
  - Saturation - queue length or time
  - Error - software or hardware errors
- Resource include...(refer to [Brendan's USE check list](#))
  - Computing
    - App, scheduler, memory allocator, major/minor faults, swapping, locks...
    - CPU, Memory, Interconnection (QPI)
  - Storage
    - App, file system, block layer, multi-path, raid, protocol stacks(SCSI, NVMe), disk driver
    - PCIe bus, Storage bus (SAS, NVMe...), Disk or HBA counters
  - Network
    - App, TCP, UDP, NIC driver, bonding, virtual switch...
    - PCIe bus, NIC perf counters



# Analysis - Trace Example 1

- Trace file IO size distribution to understand workload
- The [./fiohist.stp](#) is written by SystemTap scripts

```
$ sudo fio fs seq write sync_001
```

```
$ sudo ./fiohist.stp 94302
```

```
starting probe
```

```
^C
```

```
IO Summary:
```

		read	read		write	write	
name	open	read	KB tot	B avg	write	KB tot	B avg
fio	7917	0	0	0	3698312	14793248	4096

```
Write I/O size (bytes):
```

```
process name: fio
value |----- count
      1024 |          0
      2048 |          0
      4096 |@@@@@@@@@@@ 3698312
      8192 |          0
     16384 |          0
```



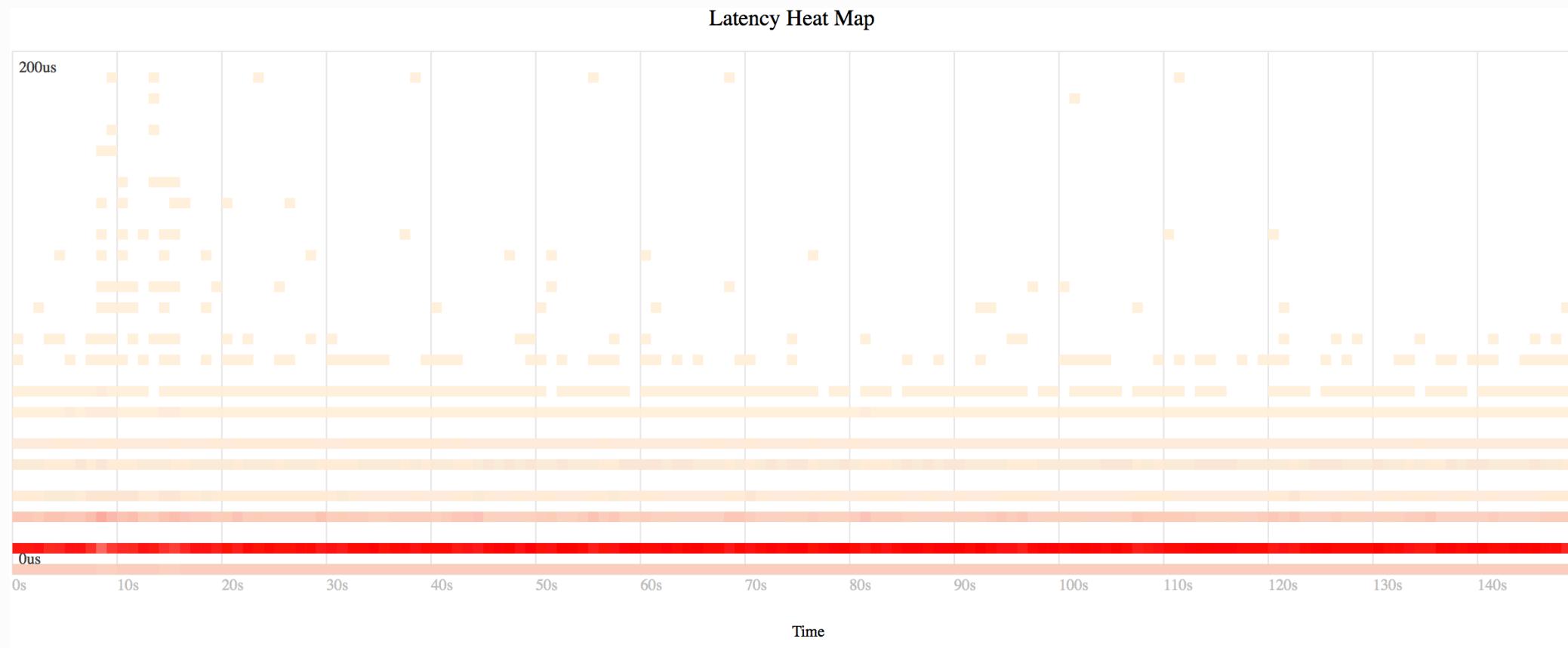
# Analysis - Trace Example 2

- Block IO latency analysis to understand the workload
- The iosnoop is based on Ftrace and written by bash & awk (perf-tools project)  
\$ sudo /ws/perf-tools/iosnoop -d 253,1 -s -t  
Tracing block I/O. Ctrl-C to end.

STARTs	ENDs	COMM	PID	TYPE	DEV	BLOCK	BYTES	LATms
11165.022265	11165.022273	fio	11426	W	253,1	2486	57344	0.01
11165.022278	11165.022291	fio	11426	W	253,1	3142	130560	0.01
11165.022292	11165.022305	fio	11426	W	253,1	3397	130560	0.01
11165.022305	11165.022319	fio	11426	W	253,1	3652	130560	0.01
11165.022319	11165.022333	fio	11426	W	253,1	3907	130560	0.01
11165.022333	11165.022347	fio	11426	W	253,1	4162	130560	0.01
11165.022347	11165.022361	fio	11426	W	253,1	4417	130560	0.01
11165.022362	11165.022375	fio	11426	W	253,1	4672	130560	0.01
11165.022375	11165.022388	fio	11426	W	253,1	4927	130560	0.01



# Analysis - Heatmap

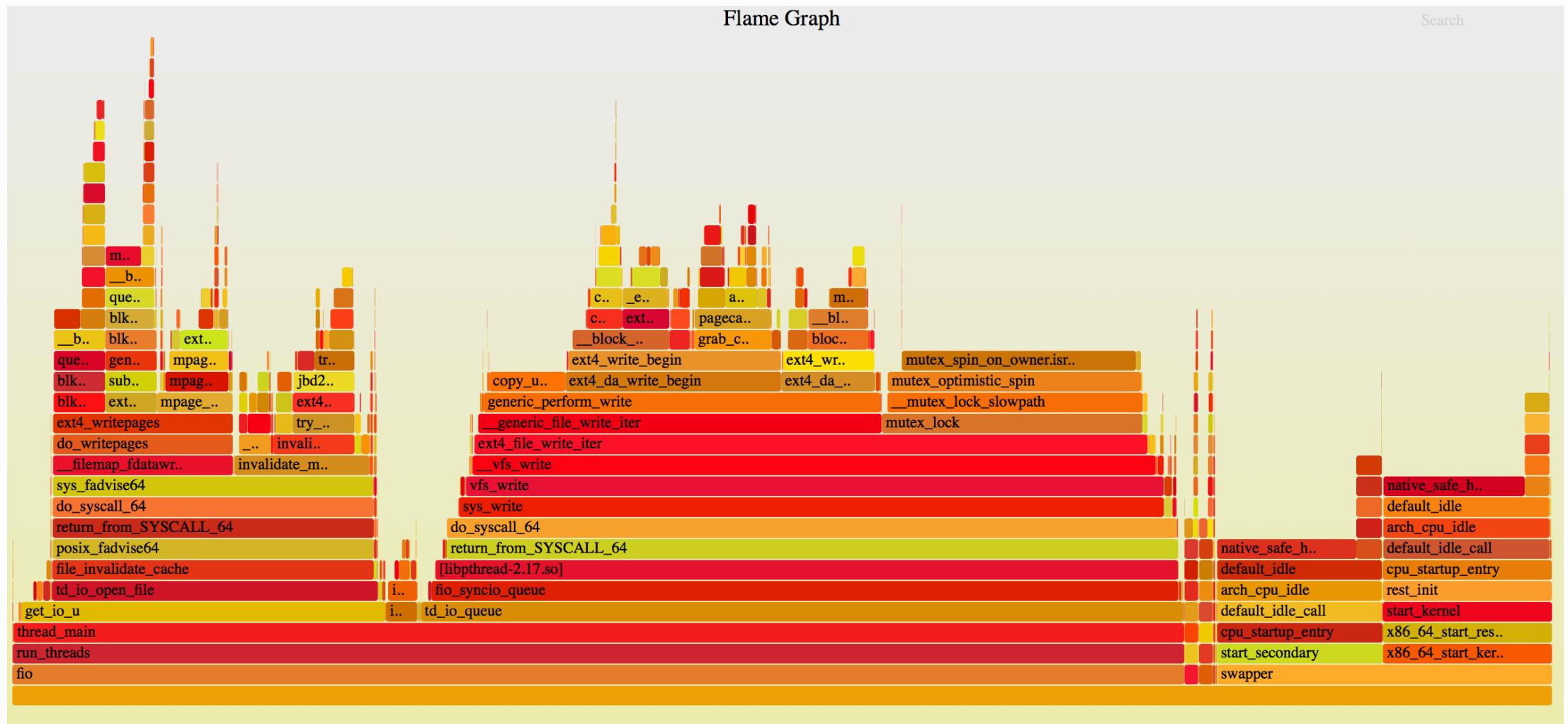


# Analysis - Profiling Example

- Run FIO with [write sequential IO test](#)
- Sampling all applications and kernel activities on CPU  
\$ sudo perf record -a -g --call-graph dwarf -F 997 sleep 60  
\$ sudo perf report > [perf\\_record\\_fs\\_seq\\_write\\_sync\\_001.log](#)
- Virtualize profiling data by Flamegraph,
  - sudo perf script | stackcollapse-perf.pl > out.perf-folded
  - cat out.perf-folded | flamegraph.pl > [perf-kernel.svg](#)



# Analysis - Flamegraph



# System Behaviors Learning

- Learn runtime code path
  - Function pointers
  - Multiple code paths
  - Chain of stack backtraces
- Understand system behaviors under real workload
  - Fast vs. slow code path
  - Identify hot code path
- Present trace logs
  - Backtrace
  - Function graph
  - Statistic based on stack backtrace



# Learning - Backtrace Example

- Understand block driver run time code path under real work load

```
# stap --all-modules -e 'probe module("sampleblk").function("sampleblk_request") {  
print_backtrace() print("===\n") }'
```

```
0xfffffffffa04a1030 : sampleblk_request+0x0/0x0 [sampleblk]  
0xffffffff81313793 : __blk_run_queue+0x33/0x40 [kernel]  
0xffffffff8131380a : queue_unplugged+0x2a/0xb0 [kernel]  
0xffffffff81318e81 : blk_flush_plug_list+0x1d1/0x220 [kernel]  
0xffffffff8131928c : blk_finish_plug+0x2c/0x40 [kernel]  
0xfffffffffa0729590 : ext4_writepages+0x4e0/0xcf0 [ext4]  
0xffffffff8119cd3e : do_writepages+0x1e/0x30 [kernel]  
0xffffffff81190b66 : __filemap_fdatasync_range+0xc6/0x100 [kernel]  
0xffffffff811d5d96 : SyS_fadvise64+0x216/0x250 [kernel]  
0xffffffff81003c12 : do_syscall_64+0x62/0x110 [kernel]  
0xffffffff816bb721 : return_from_SYSCALL_64+0x0/0x6a [kernel]
```

====



# Learning – Function Graph Example

- Backtrace vs. Function Graph
  - Bottom to top vs. top to bottom
  - Direct code path, vs. full code path coverage
  - Less tracing overhead vs. bigger overhead
- Understand fadvise64 syscall behaviors

```
$ sudo ./funcgraph -d 1 -p 95069 SyS_fadvise64 > functiongraph.log
```

```
Tracing "SyS_fadvise64" for PID 95069 for 1 seconds...
 0)           | SyS_fadvise64() {
 0)           |   __fdget() {
 0)           |     __fget_light();
 0)           |   }
 0)           |   __raw_spin_lock();
 0)           | }
 0)           | SyS_fadvise64() {
 0)           |   __fdget() {
 0)           |     __fget_light();
 0)           |   }
 0)           |   inode_congested();
 0)           |   __filemap_fdatawrite_range() {
 0)           |     __raw_spin_lock();
 0)           |     wbc_attach_and_unlock_inode();
 0)           |     do_writepages() {
```



# Learning - Hot Path Example

```
73.77% 73.77% fio          [sampleblk]      [k] sampleblk_request
|
---sampleblk_request
    queue_unplugged
    blk_flush_plug_list
    |
    |--74.45%-- blk_finish_plug
        ext4_writepages
        do_writepages
        __filemap_fdatawrite_range
        sys_fadvise64
        do_syscall_64
        return_from_SYSCALL_64
        posix_fadvise64
        0x100000004041a08
    |
    |--25.55%-- blk_queue_bio
        generic_make_request
        submit_bio
        ext4_io_submit
        |
        |--99.95%-- ext4_writepages
            do_writepages
            __filemap_fdatawrite_range
            sys_fadvise64
            do_syscall_64
            return_from_SYSCALL_64
            posix_fadvise64
            0x100000004041a08
    |
    |--0.05%-- [...]
```



# Learning – Uprobe Example

- Under FIO benchmark behaviors
  - Question: How and why fadvise64 system call gets called?
  - Trace fio by using perf probe:
    - \$ sudo perf probe -x /lib64/libc.so.6 -F | grep fadvise64
    - \$ sudo perf probe -x /lib64/libc.so.6 -L posix\_fadvise64
    - \$ sudo perf probe -x /lib64/libc.so.6 -V posix\_fadvise64
    - \$ sudo perf probe -x /lib64/libc.so.6 -a 'posix\_fadvise64 advise'
    - \$ sudo perf record -e probe\_libc:posix\_fadvise64 -aR -g --call-graph dwarf sleep 60
    - \$ sudo perf report
    - \$ sudo perf script
  - Please [access trace logs here.](#)



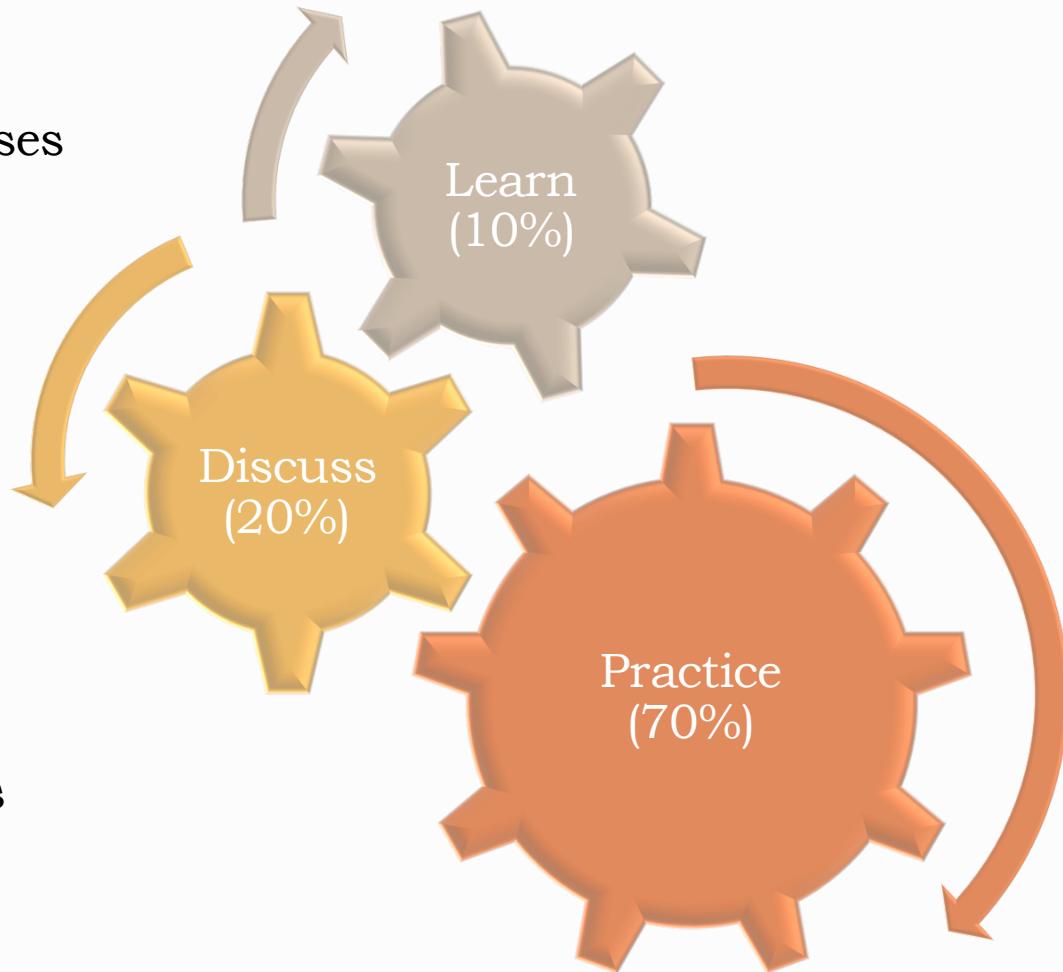
# AGENDA

- Overview
- Trace Building Blocks
- Diagnosis, Analysis and Learning
- For Fun...



# Have Fun!

- Practice first
  - Have the basic concepts about use cases
  - Use it before understand
  - Leverage toolkits and examples
- Discuss efficiently
  - Define question clearly
  - Know the knowledge dependencies
- Learn objectively
  - Have a specific goal
    - Use case: Developer vs. Admin
    - What are learning outcomes
  - Find right reference: book, documents



# Resources

- Links
  - [Brendan Gregg website](#) – many good articles
  - [Brendan's Linux Perf collections](#)
  - [Perf examples by Brendan Gregg](#)
  - [Oliver Yang's Blog](#) - blogs tagged with perf or trace
- Document
  - [Ftrace Document](#)
  - [Kprobe Event Document](#)
  - [Tracepoint Document](#)
  - [Uprobe Event Document](#)
  - [SystemTap Document Collection](#)
  - [System Methodology Holistic Performance Analysis on Modern Systems](#)
    - By Brendan Gregg, ACM Applicative System Methodology 2016.1
- Book
  - [Systems Performance: Enterprise and the Cloud](#) by Brendan Gregg



**Q & A**

**Thank You**

