Profiling JVM Applications in Production

Sasha Goldshtein
CTO, Sela Group

@goldshtn
github.com/goldshtn

https://s.sashag.net/srecon0318

Workshop Introduction

- Mission:
  Apply modern, low-overhead, production-ready tools to monitor and improve JVM application performance on Linux
- Objectives:

❑Identifying overloaded resources

❑Profiling for CPU bottlenecks

❑Visualizing and exploring stack traces using flame graphs

❑Recording system events (I/O, network, GC, etc.)

❑Profiling for heap allocations

Course Introduction

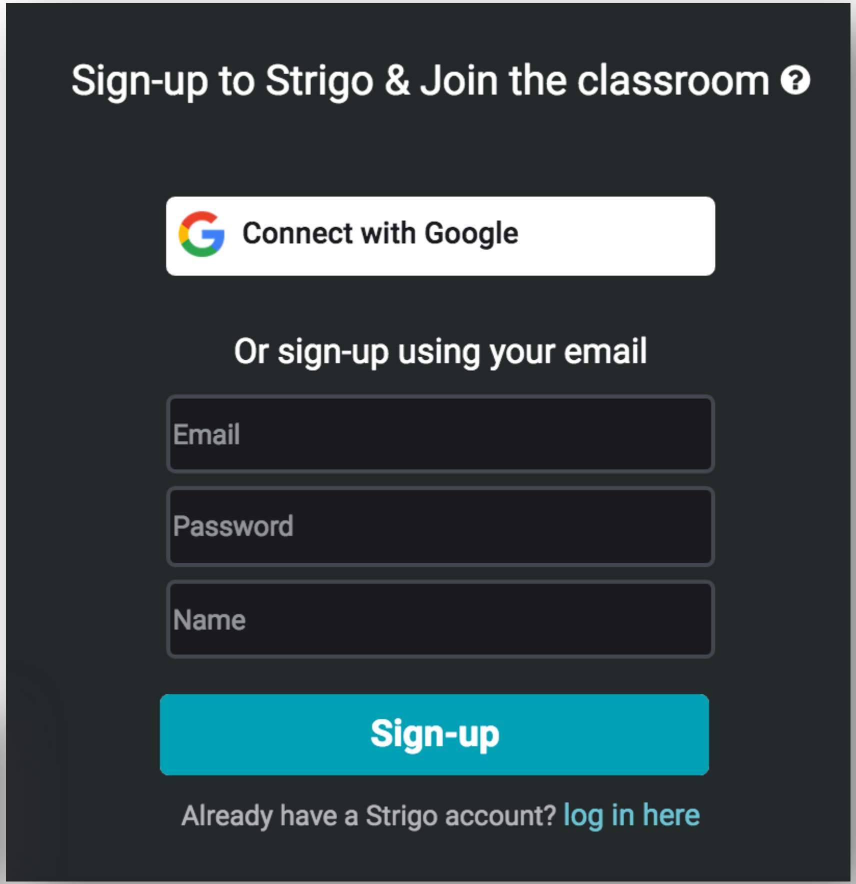
- Target audience:
  Application developers, system administrators, production engineers
- Prerequisites:
  Understanding of JVM fundamentals, experience with Linux system administration, familiarity with OS concepts
- Lab environment:
  EC2, delivered through the browser during the course dates
- Course hands-on labs:
  https://github.com/goldshtn/linux-tracing-workshop

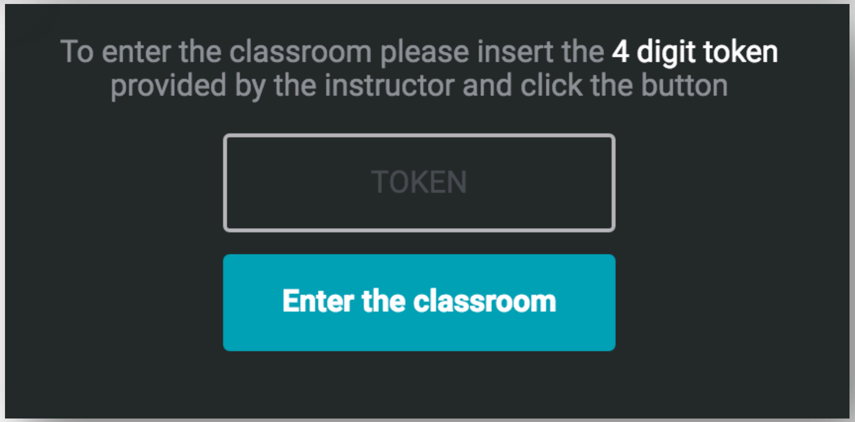
Course Plan

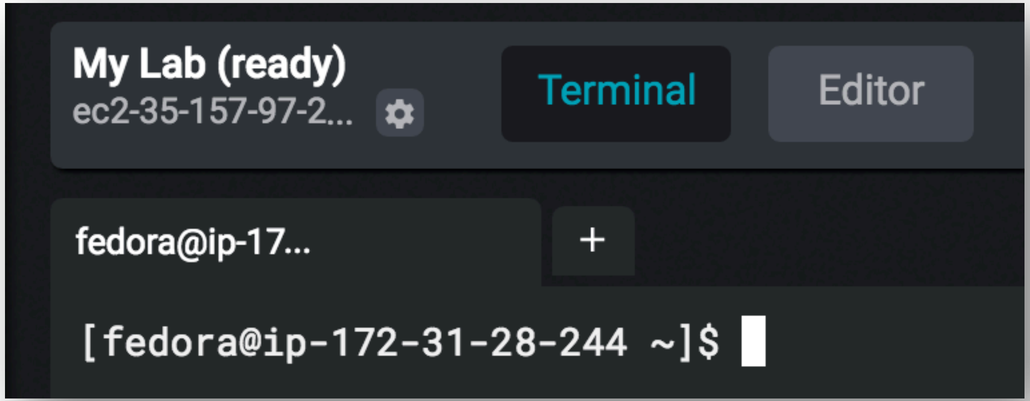
- JVM and Linux performance information sources
- CPU sampling
- Flame graphs and symbols
- **Lab**: Profiling with perf and async-profiler
- eBPF
- BCC tools
- **Lab**: Tracing file opens
- GC tracing and allocation profiling
- **Lab**: Allocation profiling

The Lab Environment

- Follow the link provided by the instructor
- Sign up or log in with Google
- Enter the classroom token
- Click the beaker-in-a-cloud icon to get your own lab instance
- Wait for the terminal to initialize

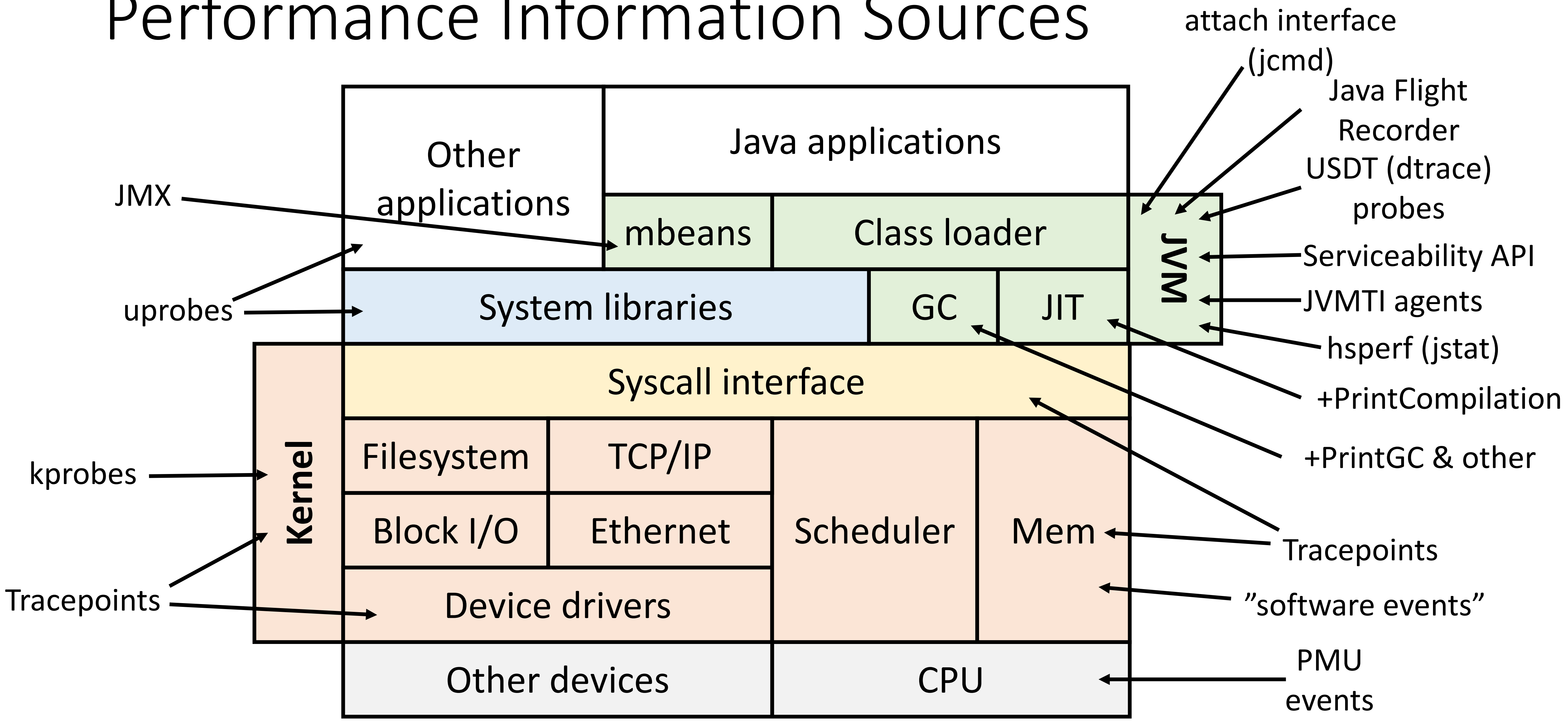






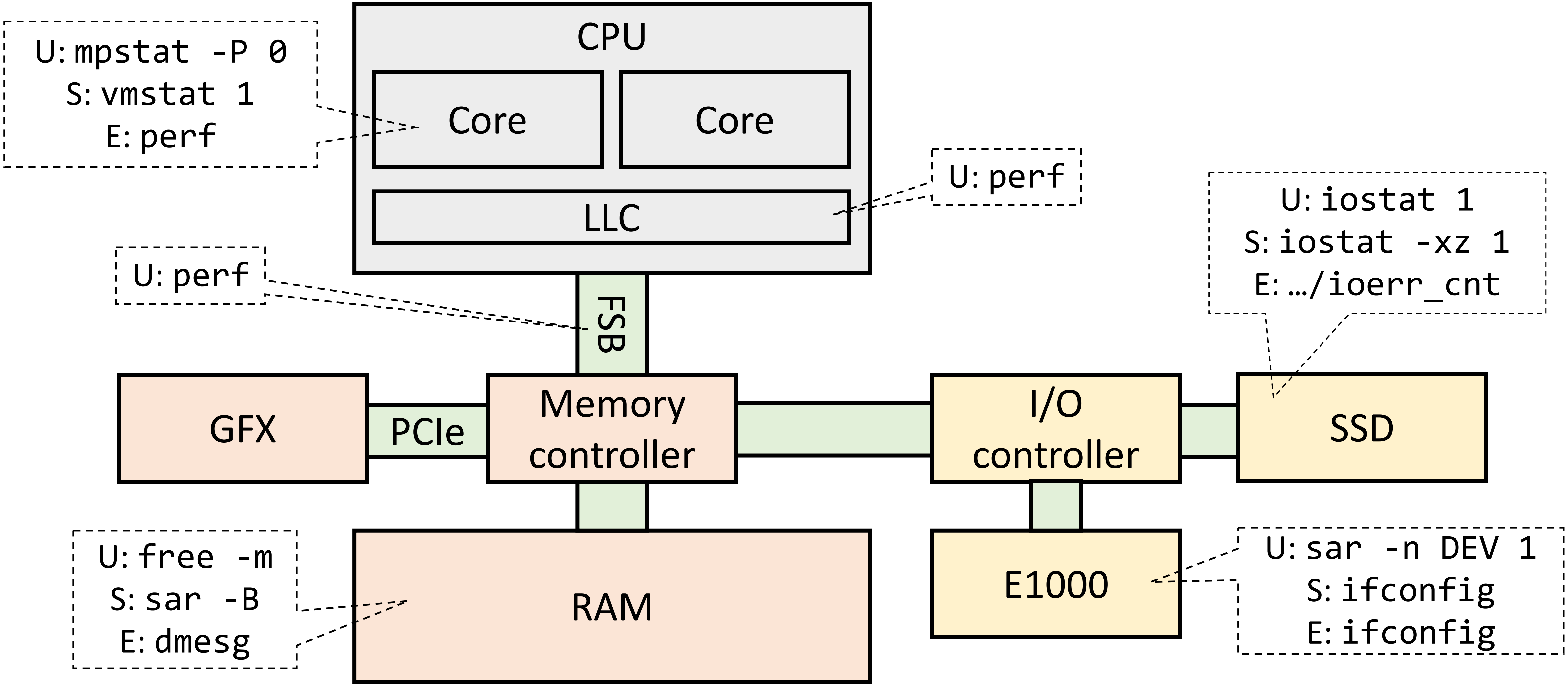
JVM and Linux Performance Sources

Performance Information Sources

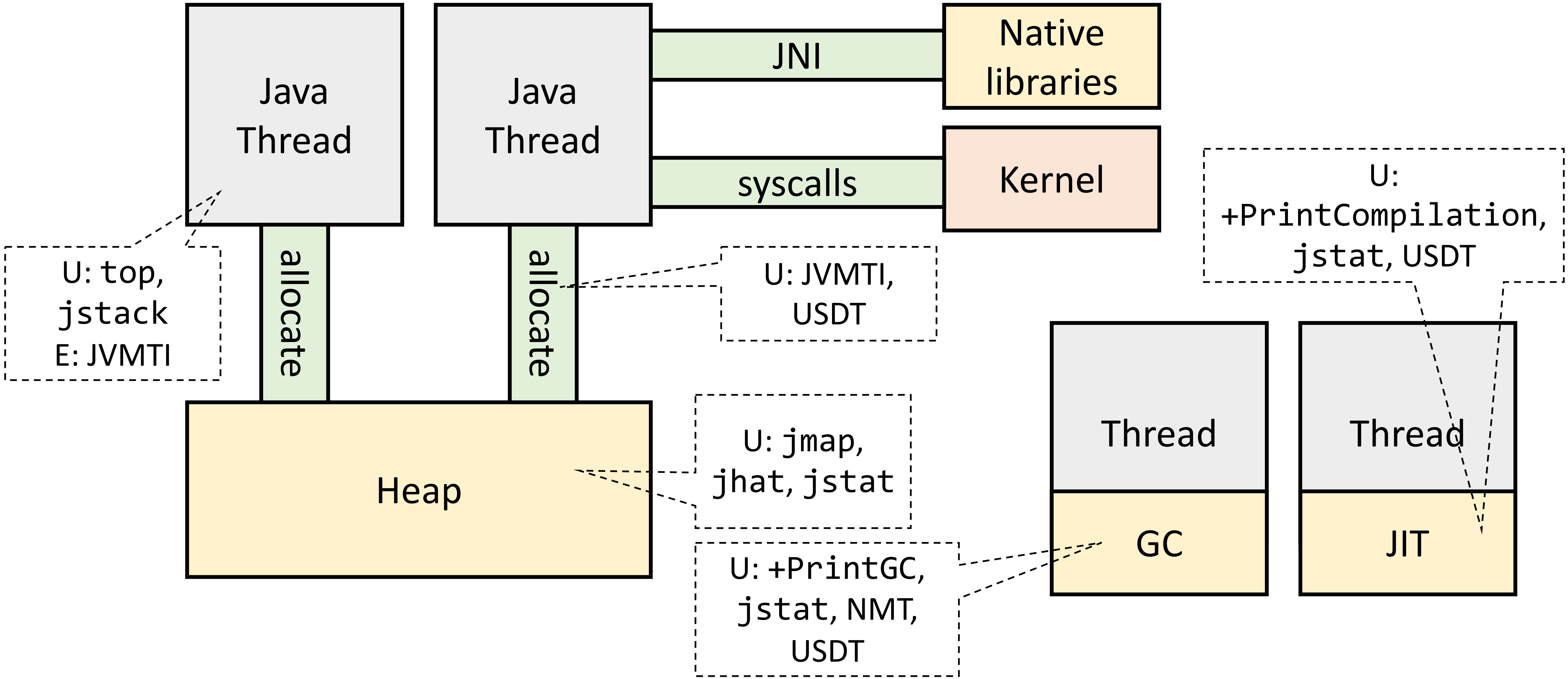


USE Checklist for Linux Systems

http://www.brendangregg.com/USEmethod/use-linux.html



USE Checklist For JVM Applications



⚠️ Mind The Overhead

- Any observation can change the state of the system, but some observations are worse than others
- Performance tools have overhead
  - Check the docs
  - Try on a test system first
  - Measure degradation introduced by the tool

OVERHEAD

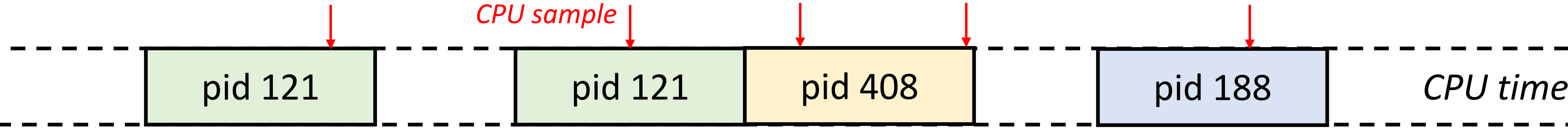
This traces various kernel page cache functions and maintains in-kernel counts, which are asynchronously copied to user-space. While the rate of operations can be very high (>1G/sec) we can have up to 34% overhead, this is still a relatively efficient way to trace these events, and so the overhead is expected to be small for normal workloads. Measure in a test environment.

—*`man cachestat`* *(from BCC)*

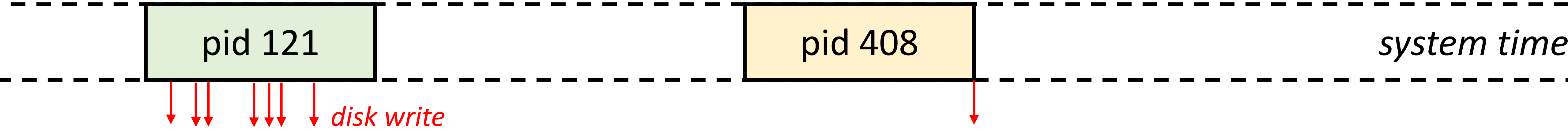
CPU Sampling

Sampling vs. Tracing

- **Sampling** works by getting a snapshot or a call stack every N occurrences of an interesting event
  - For most events, implemented in the PMU using overflow counters and interrupts

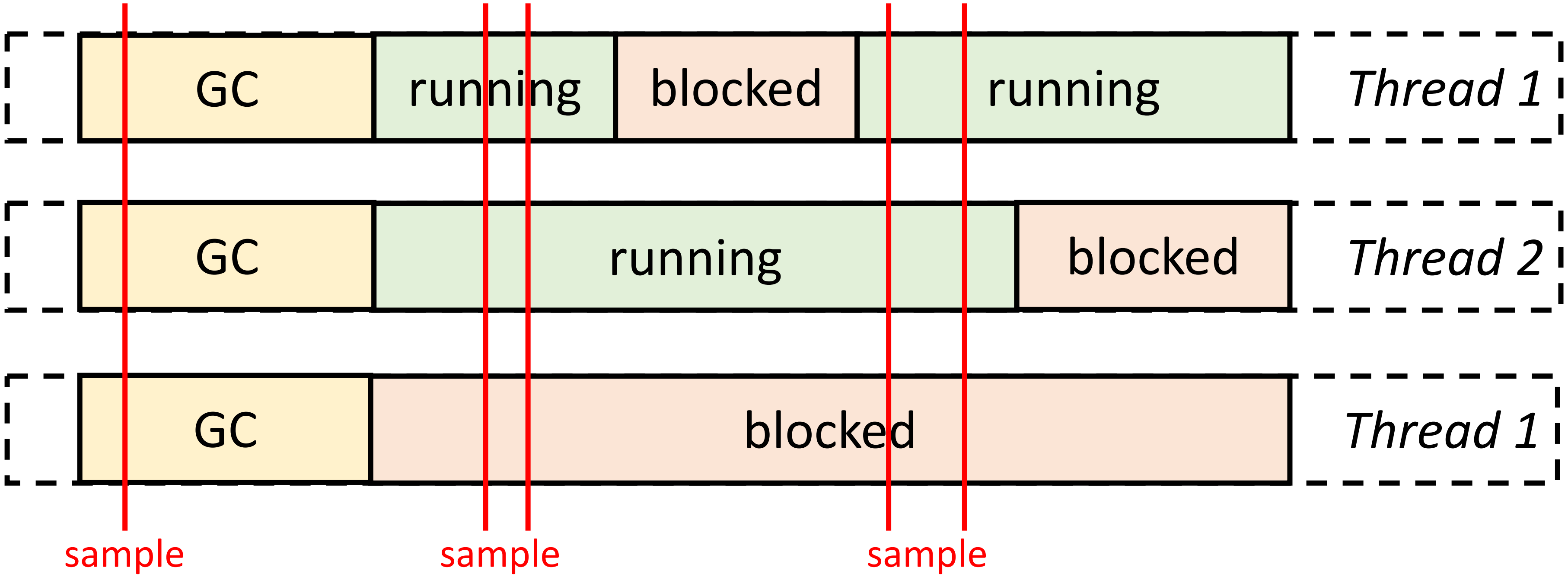


- **Tracing** works by getting a message or a call stack at every occurrence of an interesting event



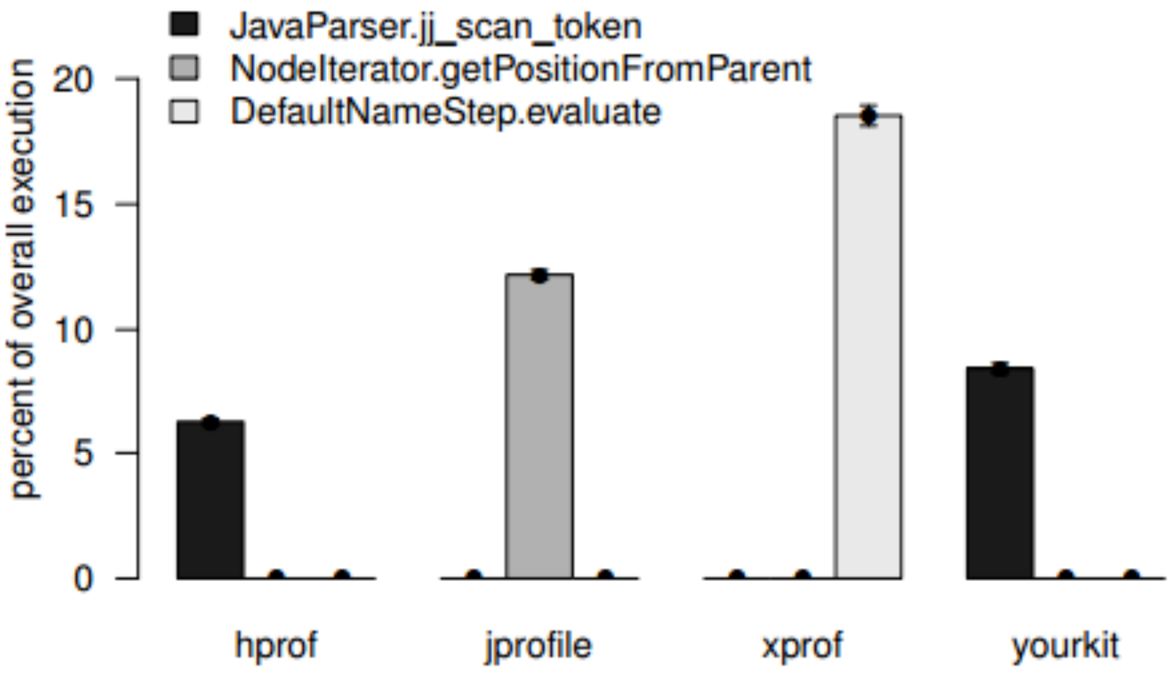
JVM Stack Sampling

- Traditional CPU profilers sample all thread stacks periodically (e.g. 100 times per second)
  - Typically use the JVMTI `GetAllStackTraces` API
  - `jstack`, JVisualVM, YourKit, JProfiler, and a lot of others



Safepoint Bias

- Samples are captured only at *safepoints*
- Research Evaluating The Accuracy of Java Profilers by Mytkowicz, Diwan, Hauswirth, Sweeney shows wild variety of results between profilers due to safepoint bias
- Additionally, capturing a full stack trace for all threads is quite expensive (think Spring)



`perf`

- `perf` is a Linux multi-tool for performance investigations
- Capable of both tracing and sampling
- Developed in the kernel tree, must match running kernel's version

- Debian-based: `apt install linux-tools-common`
- Red Hat-based: `yum install perf`

Recording CPU Stacks With `perf`

- To find a CPU bottleneck, record stacks at timed intervals:

```
# system-wide
perf record -ag -F 97

# specific process
perf record -p 188 -g -F 97

# specific workload
perf record -g -F 97 -- ./myapp
```

Legend

-a	all CPUs
-p	specific process
--	run workload and capture it
-g	capture call stacks
-F	frequency of samples (Hz)
-c	# of events in each sample

A Single Stack

```
# perf script
parprimes 13393 248974.821897:   10309278 cpu-clock:
              92b is_prime+0xffffffffff800035 (/…/parprimes)
              96c primes_loop+0xffffffffff800021 (/…/parprimes)
              9d4 primes_thread+0xffffffffff800020 (/…/parprimes)
             75ca start_thread+0xffff011d4ae720ca (/…/libpthread-2.23.so)
…
# perf script | wc –l
7214
```

Stack Report

```
# perf report --stdio
# Children      Self  Command       Shared Object       Symbol
# ........  ........  ............  ..................  ........................................
#
    72.02%    71.53%  parprimes     parprimes           [.] is_prime
            |
             --71.53%--start_thread
                       primes_thread
                       primes_loop
                       is_prime
...truncated

    27.86%     0.00%  dd            [kernel.kallsyms]   [k] vfs_read
            |
            ---vfs_read
               |
                --27.80%--__vfs_read
...truncated
```

Flame Graphs and Missing Symbols

Symbols

- `perf` needs symbols to display function names (beyond modules and addresses)
  - For compiled languages (C, Go, …) these are often embedded in the binary
  - Or installed as separate debuginfo (usually **/usr/lib/debug**)

```
$ objdump -tT /usr/bin/bash | grep readline
0000000000306bf8 g    DO .bss  0000000000000004  Base        rl_readline_state
00000000000a46c0 g    DF .text 00000000000001d4  Base        readline_internal_char
00000000000a3cc0 g    DF .text 0000000000000126  Base        readline_internal_setup
0000000000078b80 g    DF .text 0000000000000044  Base        posix_readline_initialize
00000000000a4de0 g    DF .text 0000000000000081  Base        readline
00000000003062d0 g    DO .bss  0000000000000004  Base        bash_readline_initialized
…
```

Report Without Symbols

```
# perf report --stdio
# Children      Self  Command  Shared Object       Symbol
# ........  ........  .......  ..................  .......................
#
   100.00%     0.00%  hello    hello               [.] 0xffffffffffc0051d
            |
            ---0x51d
               |
               |--54.91%--0x4f7
               |
               |--27.97%--0x4eb
               |
               |--8.73%--0x4e3
               |
                --7.97%--0x4ff
```

Java App Report

```
# perf report --stdio
# Children      Self  Command  Shared Object       Symbol
# ........  ........  .......  ..................  ......................
#
   100.00%     0.00%  java     perf-2318.map       [.] 0x00007f82b50004e7
            |
            ---0x7f82b50004e7
               |
               |--8.15%--0x7f82b510d63e
               |
               |--7.97%--0x7f82b510d6ca
               |
               |--7.07%--0x7f82b510d6c2
               |
               |--6.88%--0x7f82b510d686
               |
               |--6.16%--0x7f82b510d68e
```

perf-*PID*.map Files

- When symbols are missing in the binary, perf will look for a file named ***/tmp/perf-*PID*.map*** by default

```
$ cat /tmp/perf-1882.map
7f2cd1108880 1e8 Ljava/lang/System;::arraycopy
7f2cd1108c00 200 Ljava/lang/String;::hashCode
7f2cd1109120 2e0 Ljava/lang/String;::indexOf
7f2cd1109740 1c0 Ljava/lang/String;::charAt
…
7f2cd110ce80 120 LHello;::doStuff
7f2cd110d280 140 LHello;::fidget
7f2cd110d5c0 120 LHello;::fidget
7f2cd110d8c0 120 LHello;::fidget
…
```

Generating Map Files

- For interpreted or JIT-compiled languages, map files need to be generated at runtime
- Java: perf-map-agent
  `create-java-perf-map.sh $(pidof java)`
  - This is a JVMTI agent that attaches on demand to the Java process
  - Additional options include `dottedclass`, `unfoldall`, `sourcepos`
  - Consider **-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints** for more accurate inline info
- Other runtimes:
  - Node: `node --perf-basic-prof-only-functions app.js`
  - Mono: `mono --jitmap ...`
  - .NET Core: `export COMPlus_PerfMapEnabled=1`

Fixed Report; Still Broken

```
# perf report --stdio
# Children      Self  Command  Shared Object        Symbol
# ........  ........  .......  ..................   ......................
#
   100.00%     0.00%  java     perf-3828.map        [.] call_stub
            |
            ---call_stub
               LHello;::fidget
…
```

Walking Stacks

- To successfully walk stacks, `perf` requires* FPO to be disabled
  - This is an optimization that uses EBP/RBP as a general-purpose register rather than a frame pointer
- C/C++: `-fno-omit-frame-pointer`
- Java: `-XX:+PreserveFramePointer` *since Java 8u60*

* *When debug information is present, perf can use libunwind and figure out FPO-enabled stacks, but not for dynamic languages*

Fixed Report

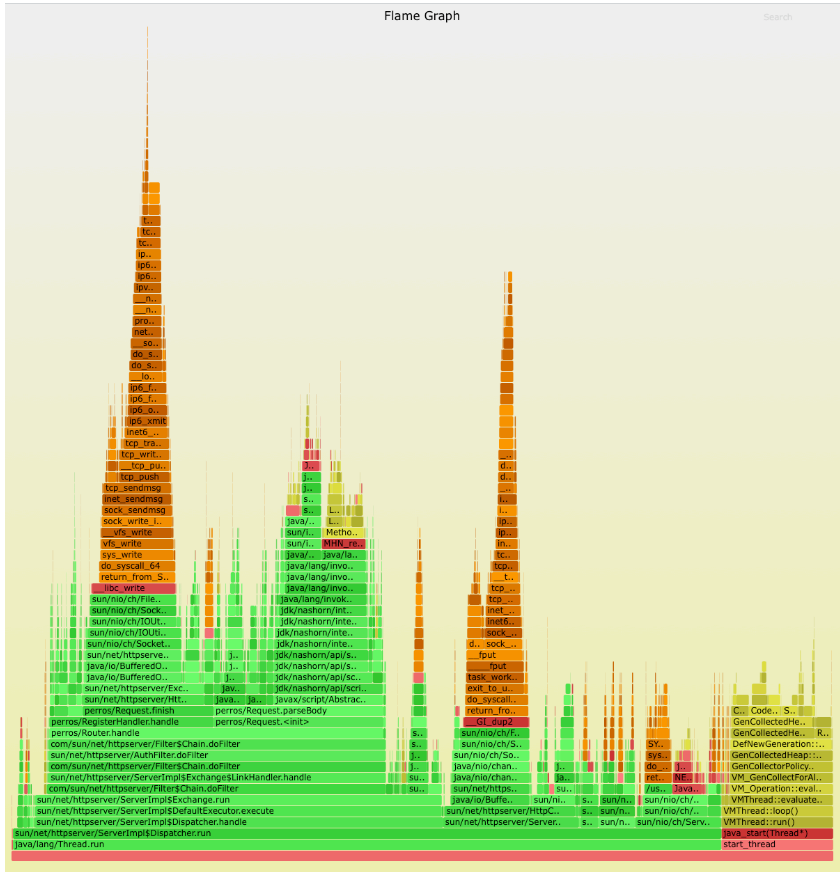
```
# perf report --stdio
# Children      Self  Command  Shared Object       Symbol
# ........  ........  .......  ..................  ......................
#
   100.00%    99.65%  java     perf-4005.map       [.] LHello;::fidget
             |
              --99.65%--start_thread
                        JavaMain
                        jni_CallStaticVoidMethod
                        jni_invoke_static
                        JavaCalls::call_helper
                        call_stub
                        LHello;::main
                        LHello;::doStuff
                        LHello;::identifyWidget
                        LHello;::fidget
…
```

Real-World Stack Reports

```
# perf report --stdio | wc -l
14823
```

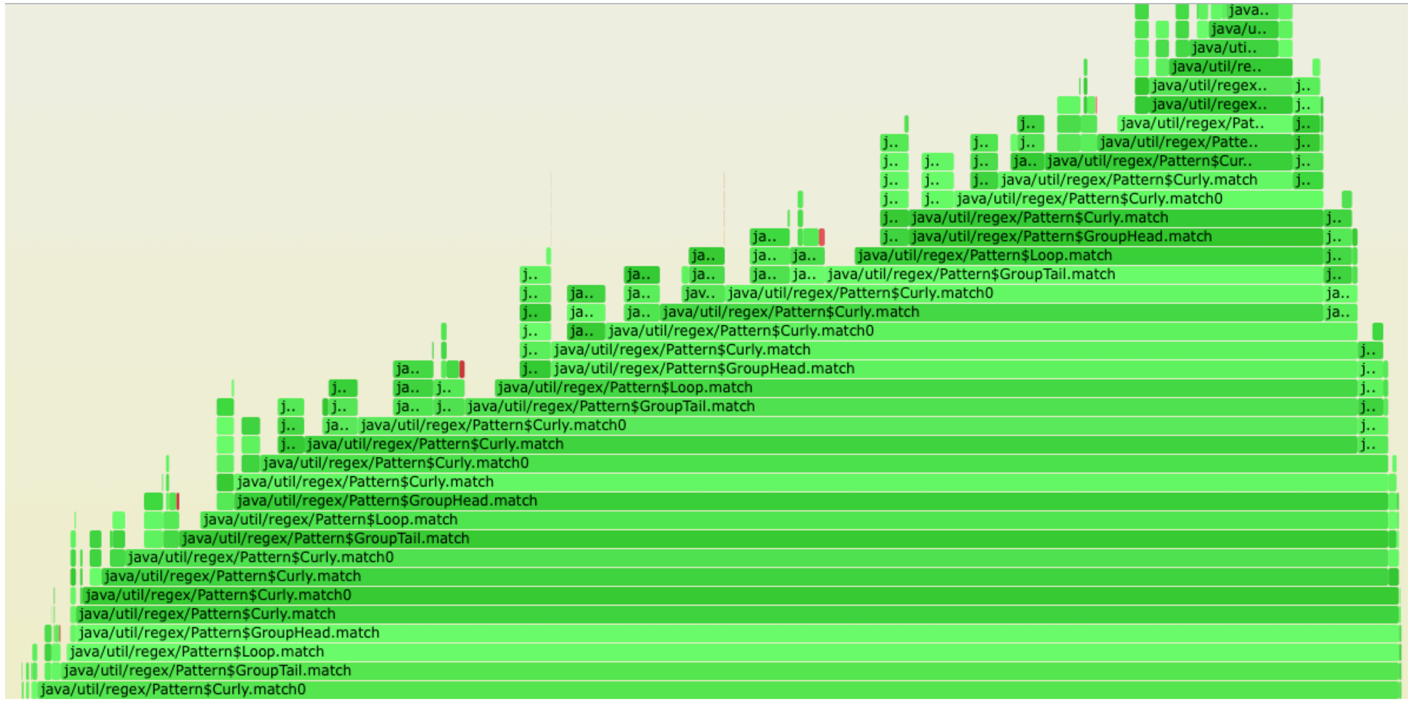
Flame Graphs

- A visualization method (adjacency graph), very useful for stack traces, invented by Brendan Gregg
  - http://www.brendangregg.com/flamegraphs.html
- Turns 1000s of stack trace pages into a single interactive graph
- Example scenarios:
  - Identify CPU hotspots on the system/application
  - Show stacks that perform heavy disk accesses
  - Find threads that block for a long time and the stack where they do it



Reading a Flame Graph

- Each rectangle is a function
- Y-axis: stack depth
- X-axis: sorted stacks (not time)
- Wider frames are more common
- Supports zoom, find
- Filter with `grep` 😎

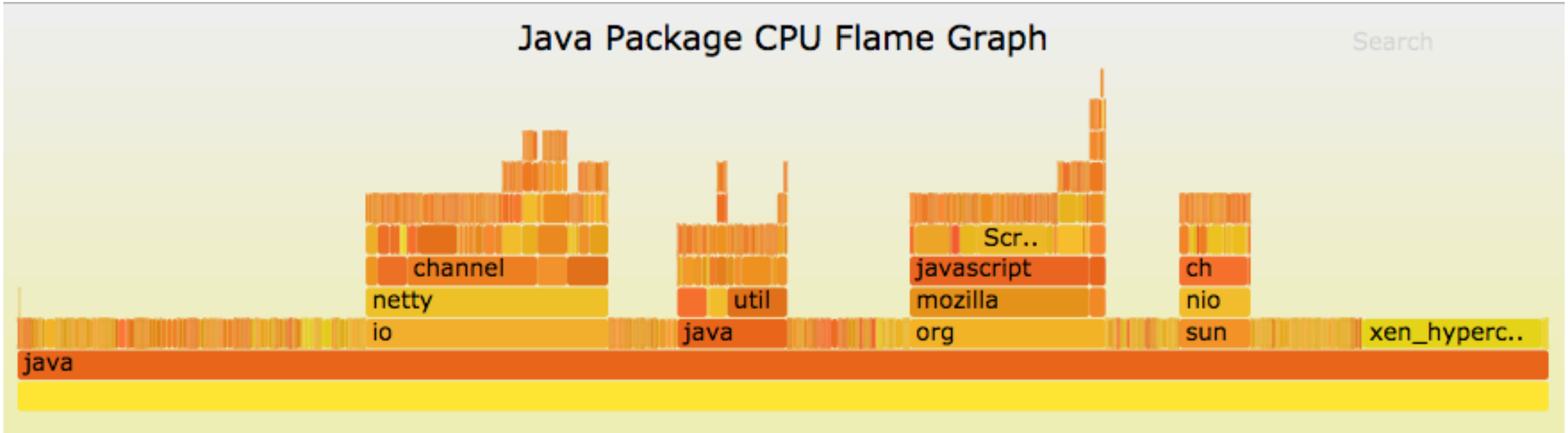


Generating a Flame Graph

```
$ git clone https://github.com/BrendanGregg/FlameGraph
$ sudo perf record -F 97 -g -p `pidof java` -- sleep 10
$ sudo perf script                           |
    FlameGraph/stackcollapse-perf.pl |
    FlameGraph/flamegraph.pl         > flame.svg
```

Not Just For Methods

- For just a package-level understanding of where your time goes, use **pkgsplit-perf.pl** and generate a package-level flame graph:



From http://www.brendangregg.com/blog/2017-06-30/package-flame-graph.html

Lab: CPU Investigation With `perf` And Flame Graphs

Problems with `perf`

- Only Java 8u60 and later is supported (to disable FPO)
- Disabling FPO has a small performance impact (up to 10% in pathological cases)
- Symbol resolution requires an additional agent
- Interpreter frames can't be resolved (shown as "Interpreter")
- Recompiled methods can be misreported (appear more than once in the perf map)
- Stack depth is usually limited to 127 (again, think Spring)
  - Can be configured since Linux 4.8 using **/proc/sys/kernel/perf_event_max_stack**

async-profiler

JVMTI Agents

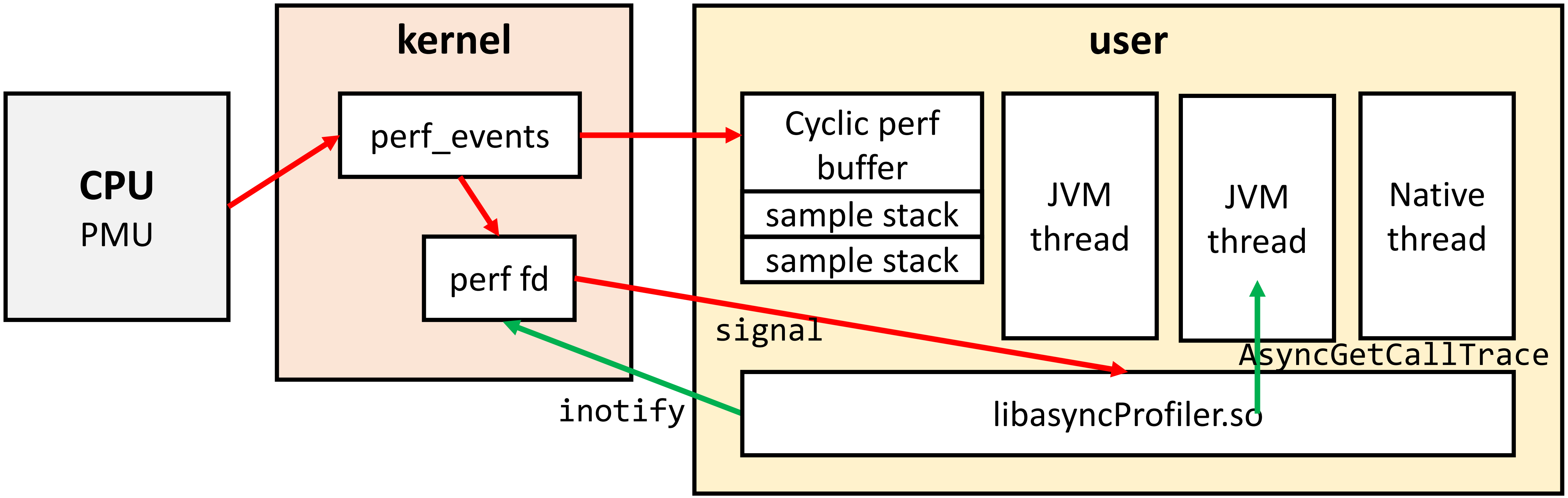
- A JVMTI (JVM Tool Interface) agent can be loaded with `-agentpath` or attached through the JVM attach interface
- Examples of functionality:
  - Trace thread start and stop events
  - Count monitor contentions and wait times
  - Aggregate class load and unload information
  - Full event reference: http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

AsyncGetCallTrace

- Internal API introduced to support lightweight profiling in Oracle Developer Studio
- Produces a single thread's stack without waiting for a safepoint
- Designed to be called from a signal handler
- Used by Honest Profiler (by Richard Warburton and contributors): https://github.com/jvm-profiling-tools/honest-profiler

async-profiler

- Open source profiler by Andrei Pangin and contributors: https://github.com/jvm-profiling-tools/async-profiler



Profilers, Compared

perf

- Java ≧8u60 to disable FPO
- Disabling FPO has a perf penalty
- Need a map file
- Interpreter frames are <u>not</u> supported
- System-wide profiling is possible
- Can profile containers from the host (or from a sidecar)

async-profiler

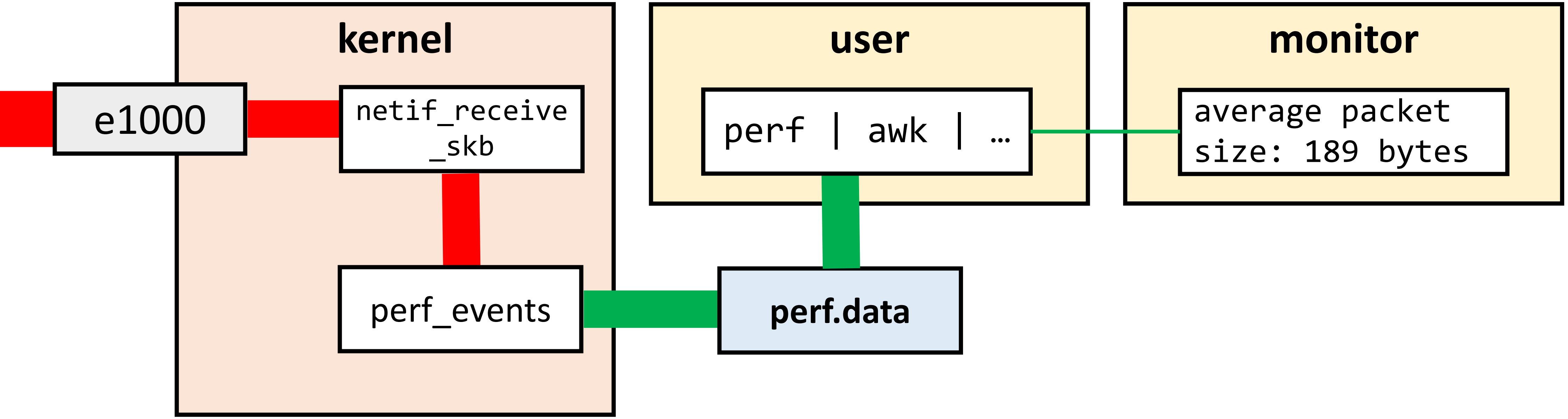
- Works on older Java versions
- FPO can stay on
- No map file is required
- Interpreter frames are supported
- In theory, native and Java stacks don't always sync
- Profiling runs in-process (so, in-container)

Lab: Profiling With **async-profiler**

eBPF

What's Wrong With `perf`?

- `perf` relies on pushing a *lot of data* to user space, through *files*, for *analysis*
  - Downloading a file at ~1Gb/s produces ~89K `netif_receive_skb` events/s (19MB/s including stacks)



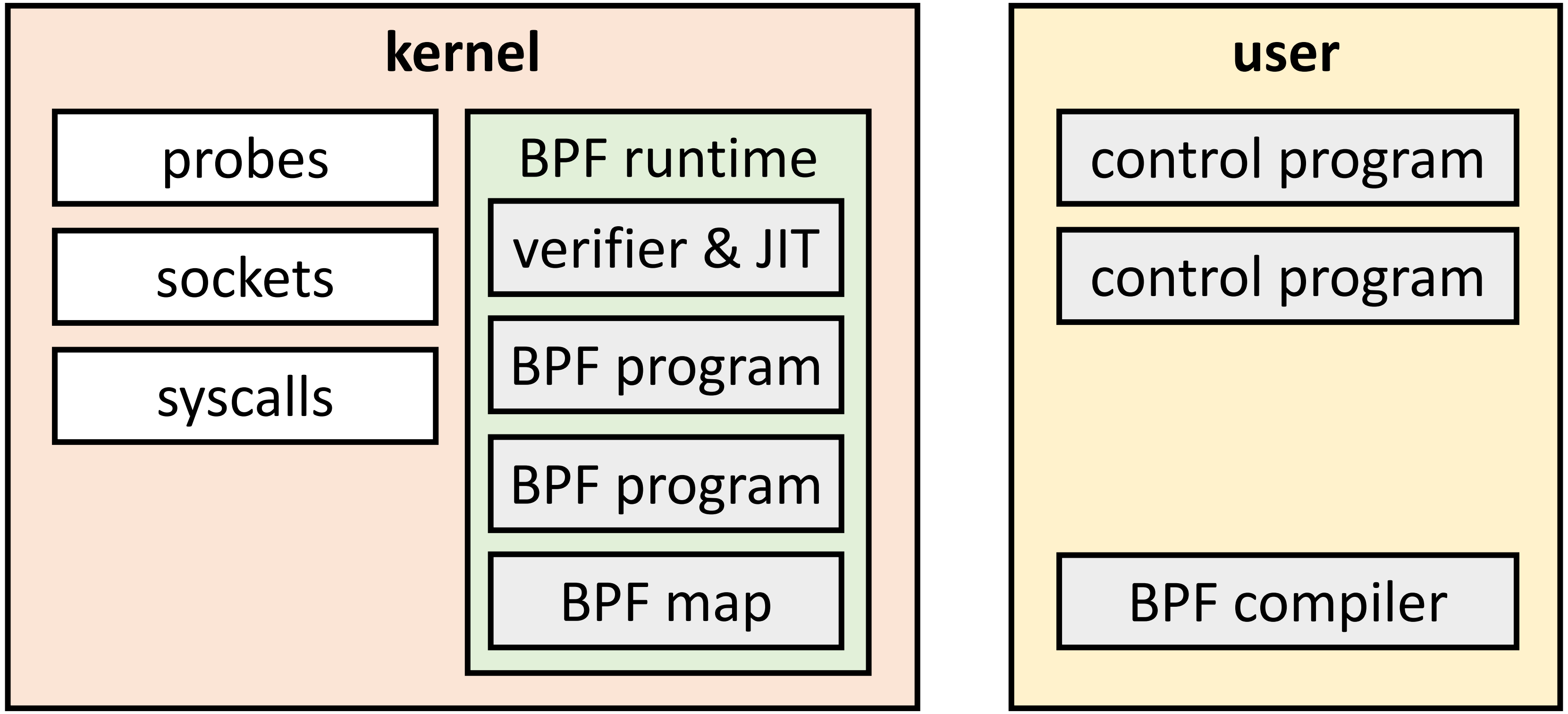
BPF: 1990

- Invented by McCanne and Jacobson at Berkeley, 1990-1992: instruction set, representation, implementation of packet filters

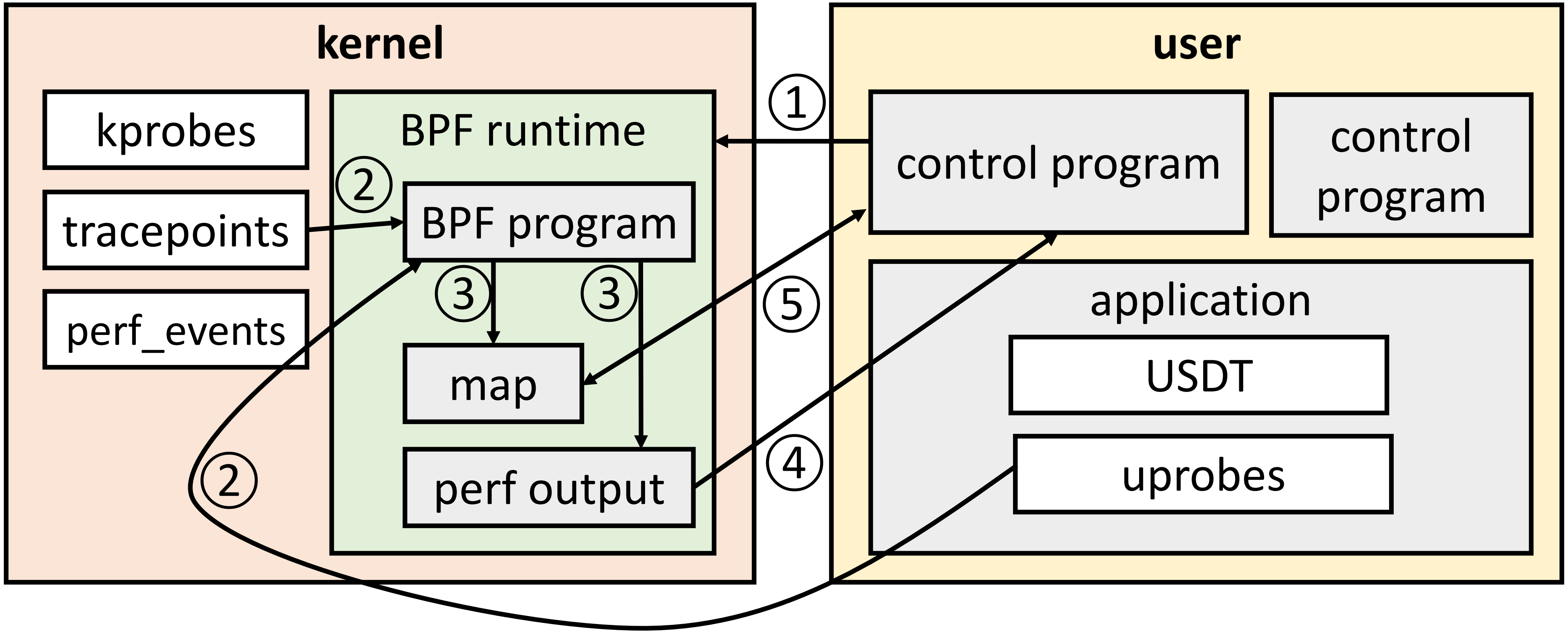
```
$ tcpdump -d 'ip and dst 186.173.190.239'
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 5
(002) ld       [30]
(003) jeq      #0xbaadbeef      jt 4    jf 5
(004) ret      #262144
(005) ret      #0
```

BPF: Today

- Supports a wide spectrum of usages
- Has a JIT for maximum efficiency



BPF Tracing



① installs BPF program and attaches to events
② events invoke the BPF program
③ BPF program updates a map or pushes a new event to a buffer shared with user-space

④ user-space program is invoked with data from the shared buffer
⑤ user-space program reads statistics from the map and clears it if necessary

BPF Tracing Features in The Linux Kernel

Version	Feature	Scenarios
4.1	kprobes/uprobes attach	Dynamic tracing with BPF becomes possible
4.1	`bpf_trace_printk`	BPF programs can print output to ftrace pipe
4.3	perf_events output	Efficient tracing of large amounts of data for analysis in user-space
4.6	Stack traces	Efficient aggregation of call stacks for profiling or tracing
4.7	Tracepoints support	API stability for tracing programs
4.9	perf_events attach	Low-overhead profiling and PMU sampling

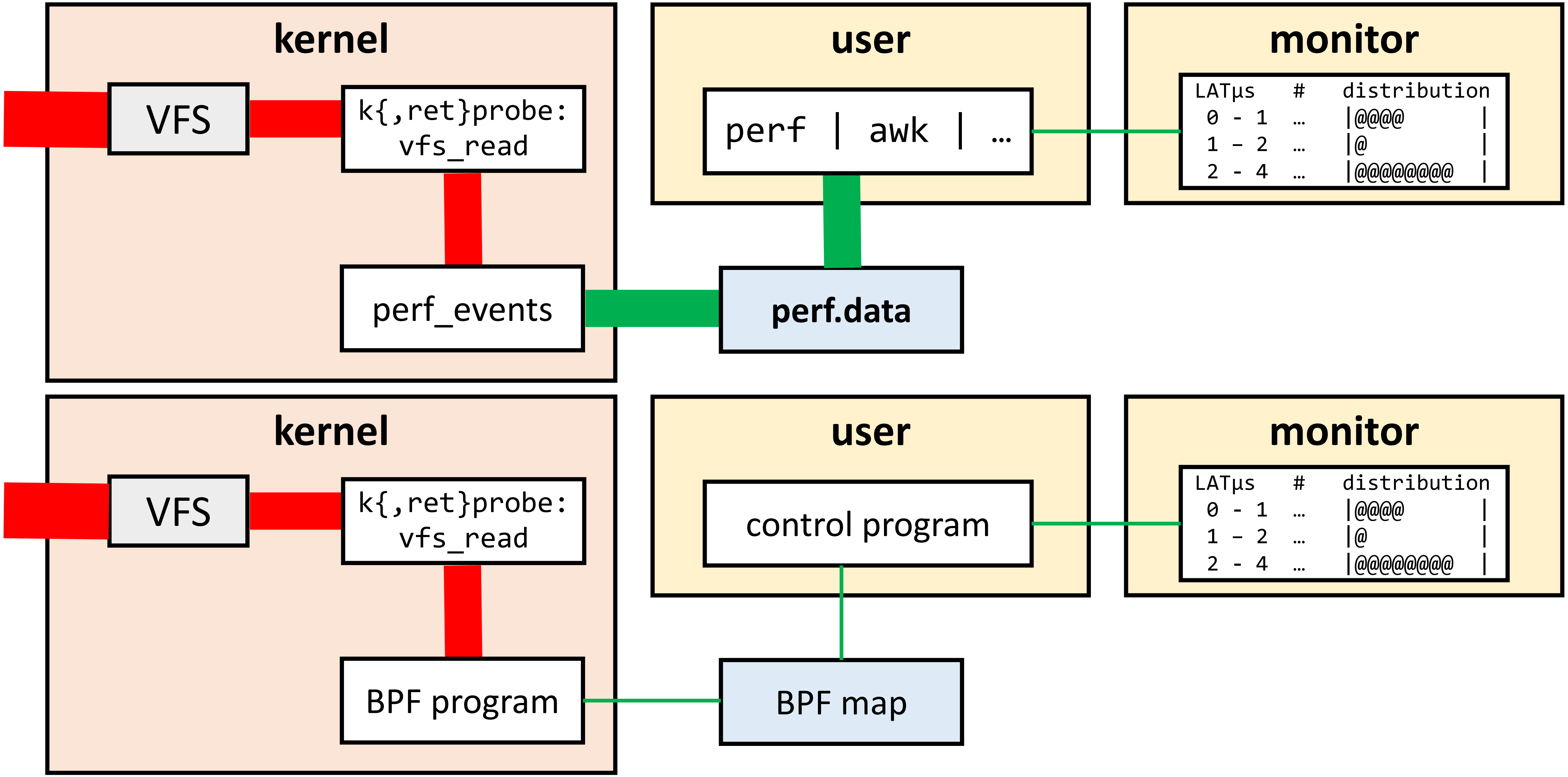
24

16.04

25

16.10

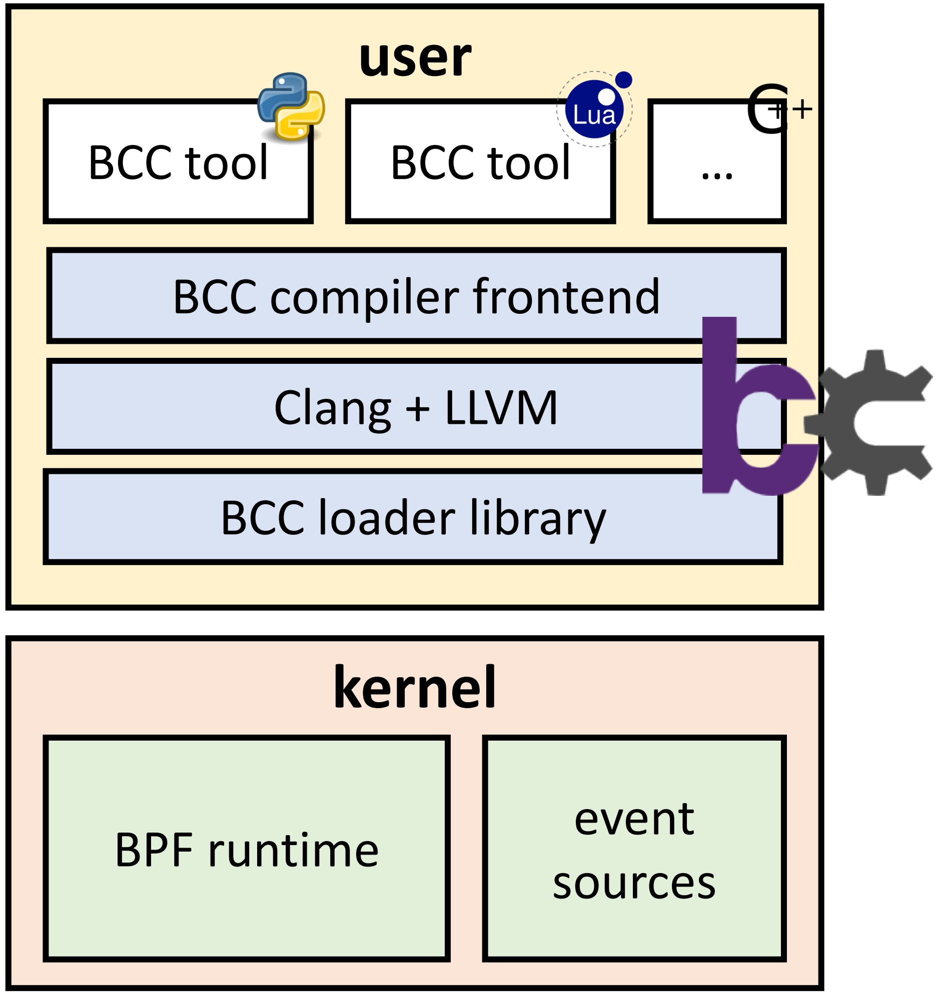
The Old Way And The New Way

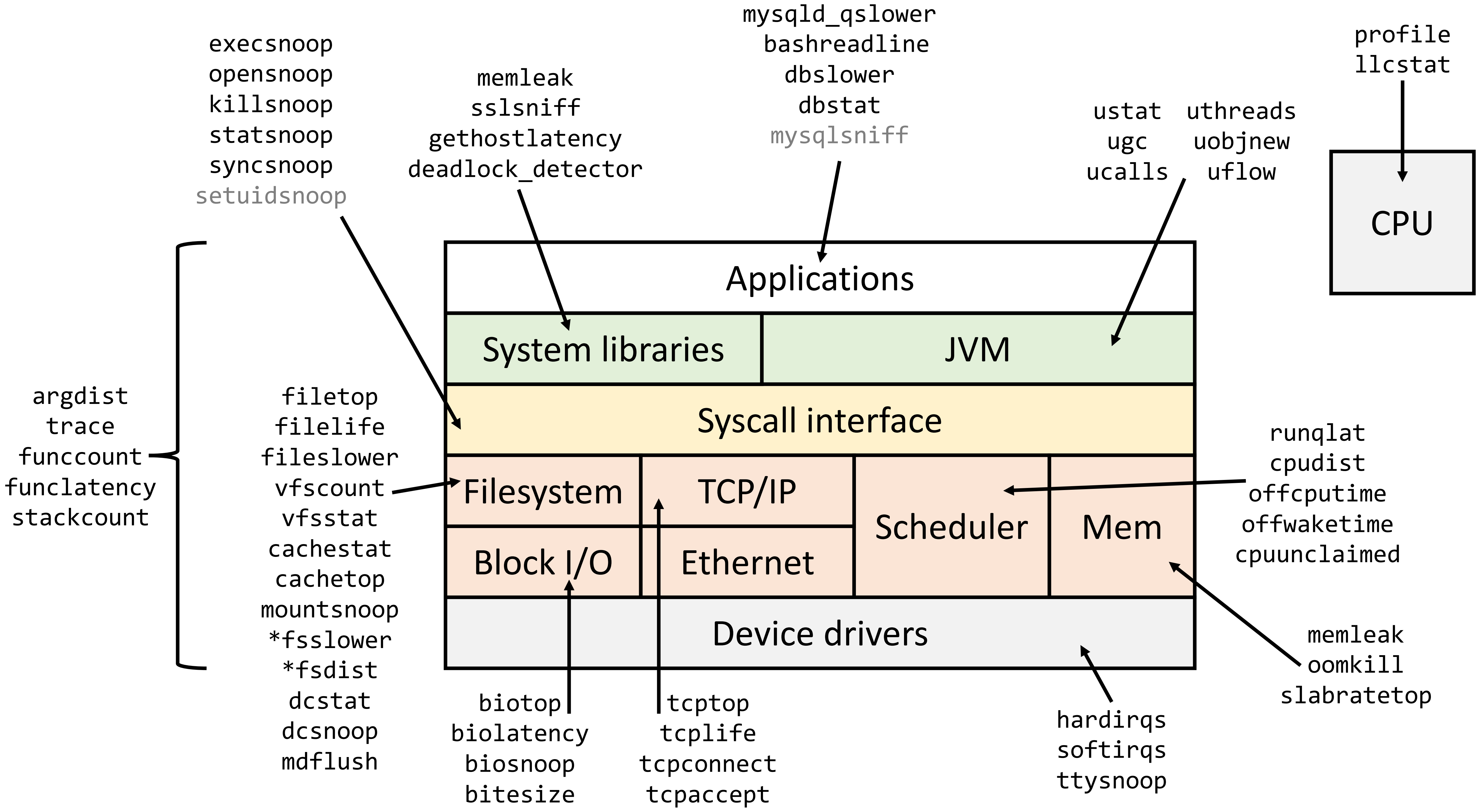


BCC Performance Checklist

The BCC BPF Front-End

- https://github.com/iovisor/bcc
- BPF Compiler Collection (BCC) is a BPF frontend library and a massive collection of performance tools
  - Contributors from Facebook, PLUMgrid, Netflix, Sela
- Helps build BPF-based tools in high-level languages
  - Python, Lua, C++





execsnoop
opensnoop
killsnoop
statsnoop
syncsnoop
setuidsnoop
memleak
sslsniff
gethostlatency
deadlock_detector
mysqld_qslower
bashreadline
dbslower
dbstat
mysqlsniff
ustat
ugc
ucalls
uthreads
uobjnew
uflow
profile
llcstat
CPU
Applications
System libraries
JVM
Syscall interface
Filesystem
TCP/IP
Scheduler
Mem
Block I/O
Ethernet
Device drivers
argdist
trace
funccount
funclatency
stackcount
filetop
filelife
fileslower
vfscount
vfsstat
cachestat
cachetop
mountsnoop
*fsslower
*fsdist
dcstat
dcsnoop
mdflush
runqlat
cpudist
offcputime
offwaketime
cpuunclaimed
memleak
oomkill
slabratetop
biotop
biolatency
biosnoop
bitesize
tcptop
tcplife
tcpconnect
tcpaccept
hardirqs
softirqs
ttysnoop

BCC Linux Performance Checklist

1. execsnoop
2. opensnoop
3. ext4slower
   (or btrfs*, xfs*, zfs*)
4. biolatency
5. biosnoop
6. cachestat
7. tcpconnect
8. tcpaccept
9. tcptop
10. gethostlatency
11. cpudist
12. runqlat
13. profile

Some BCC Tools

```
# ext4slower 1
Tracing ext4 operations slower than 1 ms
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
06:49:17 bash           3616   R 128     0           7.75 cksum
06:49:17 cksum          3616   R 39552   0           1.34 [
06:49:17 cksum          3616   R 96      0           5.36 2to3-2.7
06:49:17 cksum          3616   R 96      0          14.94 2to3-3.4
^C
```

```
# execsnoop
PCOMM            PID    RET ARGS
bash             15887    0 /usr/bin/man ls
preconv          15894    0 /usr/bin/preconv -e UTF-8
man              15896    0 /usr/bin/tbl
man              15897    0 /usr/bin/nroff -mandoc -rLL=169n -rLT=169n -Tutf8
^C
```

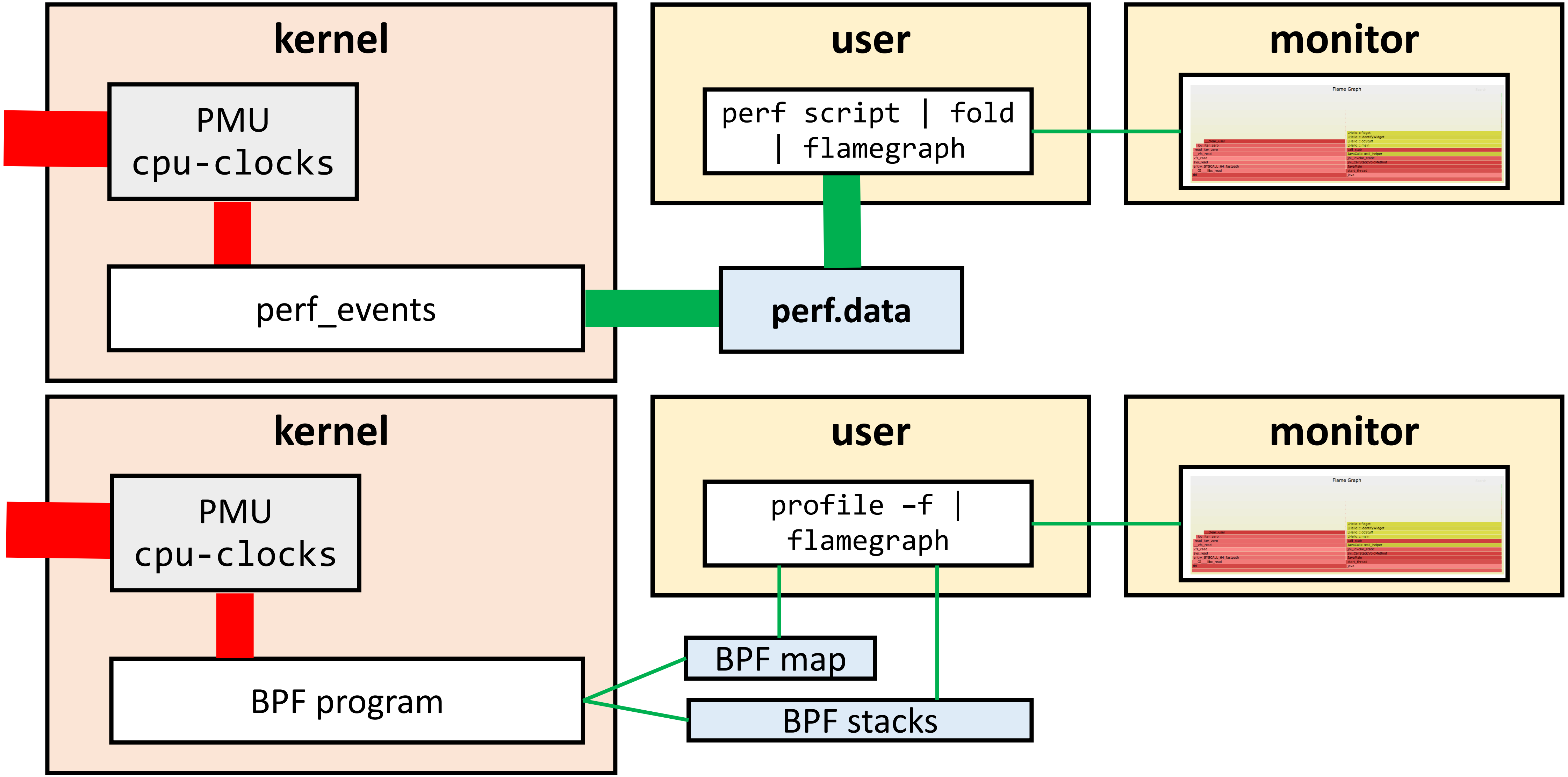
Some BCC Tools

```
# runqlat -p `pidof java` 10 1
Tracing run queue latency... Hit Ctrl-C to end.
     usecs               : count     distribution
         0 -> 1          : 11       |*                                       |
         2 -> 3          : 7        |                                        |
         4 -> 7          : 133      |******************                      |
         8 -> 15         : 288      |****************************************|
        16 -> 31         : 205      |****************************            |
        32 -> 63         : 38       |*****                                   |
        64 -> 127        : 11       |*                                       |
       128 -> 255        : 5        |                                        |
       256 -> 511        : 3        |                                        |
       512 -> 1023       : 1        |                                        |
      1024 -> 2047       : 3        |                                        |
      2048 -> 4095       : 0        |                                        |
      4096 -> 8191       : 3        |                                        |
```

BCC's profile Tool

```
# profile 10 -F 97 -K                         # kernel stacks only
…
    ffffffffa4818691 __lock_text_start
    ffffffffa45b0341 ata_scsi_queuecmd
    ffffffffa458813d scsi_dispatch_cmd
    ffffffffa458b021 scsi_request_fn
    ffffffffa43be643 __blk_run_queue
    ffffffffa43c3bc1 blk_queue_bio
    ffffffffa43c1cf2 generic_make_request
    ffffffffa43c1e4d submit_bio
    ffffffffa43b825d submit_bio_wait
    ffffffffa43c5c65 blkdev_issue_flush
    ffffffffa4309b4d ext4_sync_fs
    ffffffffa428b260 sync_fs_one_sb
    ffffffffa425a553 iterate_supers
    ffffffffa428b374 sys_sync
    ffffffffa4003c17 do_syscall_64
    ffffffffa4818bab return_from_SYSCALL_64
    -                stress (3303)
        14
```

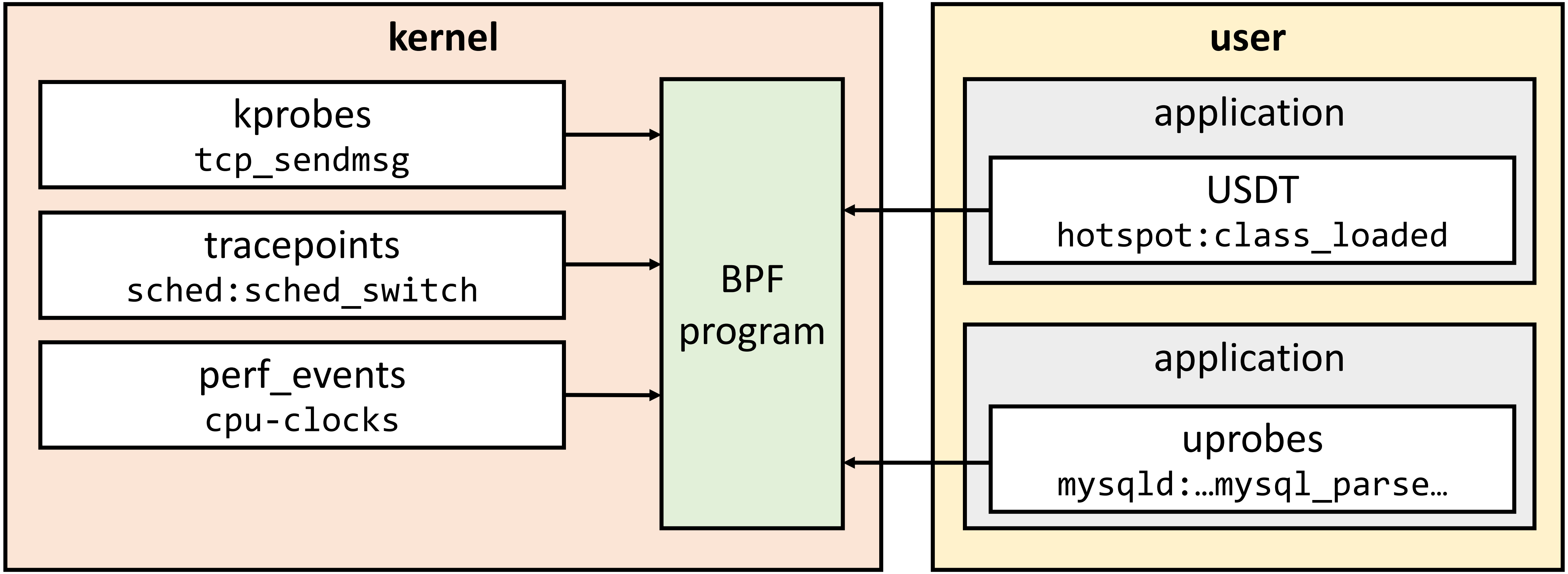
BCC's **profile** Tool



Lab: Snooping File Opens

General-Purpose BCC Tools

Tracing Sources For BCC Tools



USDT Probes in (Some) High-Level Languages

OpenJDK
hotspot:gc_begin
hotspot:thread_start
hotspot:method_entry

Oracle JDK

Node.js
node:http_server_request
node:http_client_request
node:gc_begin

libc/libpthread
libc:memory_malloc_retry
libpthread:pthread_start
libpthread:mutex_acquired

Python
python:function_entry
python:function_return
python:gc_start

Ruby
ruby:method_entry
ruby:object_create
ruby:load_entry

PHP
php:request_startup
php:function_entry
php:error

MySQL
mysql:query_start
mysql:connection_start
mysql:query_parse_start

Legend:
- OOTB
- build flag
- not supported

USDT Probes and Uprobes in the JVM

- OpenJDK Hotspot has a large number of static (USDT) probes in various subsystems; display with `tplist` or `readelf`:
```
$ tplist -p $(pidof java) | grep 'hotspot.*gc'
.../libjvm.so hotspot:mem__pool__gc__begin
.../libjvm.so hotspot:mem__pool__gc__end
.../libjvm.so hotspot:gc__begin
.../libjvm.so hotspot:gc__end
```
- All JVM native methods can be used with dynamic probes; discover with `objdump` or `nm`:
```
$ nm -C $(find /usr/lib/debug -name libjvm.so.debug)
        | grep 'card.*table'
0000000000854751 t PSScavenge::card_table()
00000000016dd778 b PSScavenge::_card_table
...
```

BCC trace

- `trace` is a multi-purpose logging tool; think of it as a dynamic log at arbitrary locations in the system (can also print call stacks)

```
# trace 'SyS_write (arg3 > 100000) "large write: %d bytes", arg3'
PID    TID    COMM            FUNC                -
9353   9353   dd              SyS_write           large write: 1048576 bytes
9353   9353   dd              SyS_write           large write: 1048576 bytes
9353   9353   dd              SyS_write           large write: 1048576 bytes
^C
```

```
# trace 'r:/usr/bin/bash:readline "%s", retval'
TIME     PID    COMM            FUNC                -
02:02:26 3711   bash            readline            ls –la
02:02:36 3711   bash            readline            wc -l src.c
^C
```

BCC `funccount/stackcount`

- `funccount` counts the number of invocations of a particular method, while `stackcount` also aggregates the call stacks

```
# LIBJVM=$(find /usr/lib -name libjvm.so)
# funccount -p $(pidof java) "$LIBJVM:*do_collection*"
Tracing 5 functions for ".../libjvm.so:*do_collection*"... Hit Ctrl-C to
end.
^C
FUNC                                              COUNT
_ZN16GenCollectedHeap13do_collectionEbbmbi      848
Detaching...
```

Lab: Tracing Database Accesses

Heap Allocation Profiling

Approaches for Allocation Profiling

- Allocation profiling can help reduce GC pressure and pause times
- Tracing each object allocation is extremely expensive, though
- Use `-XX:+ExtendedDTraceProbes` and sample `hotspot:object__alloc probes` (expect a significant overhead)
- Trace Hotspot allocation tracing callbacks designed for JFR
  - `send_allocation_in_new_tlab_event`: when a new TLAB is allocated for a thread because the old one was exhausted
  - `send_allocation_outside_tlab_event`: when an object is allocated outside a TLAB (e.g. because it’s too big, or because the TLAB is exhausted)

async-profiler

- When used with the **heap** mode, instruments the JFR TLAB allocation events and reports objects allocated and stack samples
  - Requires JDK debuginfo to be installed (to find the relevant symbols)

```
$ ./profiler.sh -d 10 -e alloc -o summary,flat `pidof java`
HEAP profiling started
...
696470120 (75.33%) [C
226075184 (24.45%) [B
   425600 (0.05%)  [Ljava/util/HashMap$Node;
   193592 (0.02%)  com/sun/org/apache/xerces/internal/dom/ElementImpl
   185536 (0.02%)  com/sun/org/apache/xml/internal/serializer/NamespaceMappings$MappingRecord
   162176 (0.02%)  java/util/Stack
```

BCC Tools With Extended Probes

```
# funccount -p `pidof java` u:$LIBJVM:object__alloc
Tracing 1 functions for "u:.../libjvm.so:object__alloc"... Hit Ctrl-C to
end.
FUNC                                                       COUNT
object__alloc                                            4000987
Detaching...

# argdist -p `pidof java` -C "u:$LIBJVM:object__alloc():char*:arg2"
        605018     arg2 = java/lang/String
        609801     arg2 = java/util/HashMap$Nod
        908716     arg2 = com/sun/org/apache/xml/internal/serializer/NamespaceMappings$MappingRecord
        908778     arg2 = java/util/Stack
        909348     arg2 = [Ljava/lang/Object;
        910097     arg2 = [C
```

grav

- Collection of performance visualization tools by Mark Price and Amir Langer: https://github.com/epickrram/grav
- Includes a Python wrapper on top of `object__alloc` probes with sampling support, flame graph generation, and filtering specific types

```
$ sudo python src/heap/heap_profile.py -p `pidof java` -d 10 > alloc.stacks
$ FlameGraph/flamegraph.pl < alloc.stacks > alloc.svg
```

Lab: Excessive GC And Allocation Profiling

Course Wrap-Up

Objectives Review

- Mission:
  Apply modern, low-overhead, production-ready tools to monitor and improve JVM application performance on Linux
- Objectives:

✓ Identifying overloaded resources

✓ Profiling for CPU bottlenecks

✓ Visualizing and exploring stack traces using flame graphs

✓ Recording system events (I/O, network, GC, etc.)

✓ Profiling for heap allocations

References

- JVM observability tools
  - http://openjdk.java.net/groups/hotspot/docs/Serviceability.html
  - http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
  - http://cr.openjdk.java.net/~minqi/6830717/raw_files/new/agent/doc/index.html
  - https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html
- `perf` and flame graphs
  - https://perf.wiki.kernel.org/index.php/Main_Page
  - http://www.brendangregg.com/flamegraphs.html
- AGCT profilers
  - https://github.com/jvm-profiling-tools/async-profiler
  - https://github.com/jvm-profiling-tools/honest-profiler
- BCC and BPF
  - https://github.com/iovisor/bcc/blob/master/docs/tutorial.md
  - http://www.brendangregg.com/ebpf.html
  - http://blogs.microsoft.co.il/sasha/2016/03/31/probing-the-jvm-with-bpfbcc/
  - http://blogs.microsoft.co.il/sasha/2016/03/30/usdt-probe-support-in-bpfbcc/
- Containers and JVM
  - https://blog.csanchez.org/2017/05/31/running-a-jvm-in-a-container-without-getting-killed/
  - http://www.brendangregg.com/blog/2017-05-15/container-performance-analysis-dockercon-2017.html
  - http://batey.info/docker-jvm-flamegraphs.html

Questions?

Sasha Goldshtein
CTO, Sela Group

@goldshtn
github.com/goldshtn