APAC Korea 2016

Linux Kernel Instrumentation in Python

Feng Li (李枫)
knewlife2015@yahoo.com

Aug 14, 2016

Agenda

I. eBPF & BCC
- eBPF
- BCC

II. In-Kernel Virtual Machine
- KPlugs
- Application

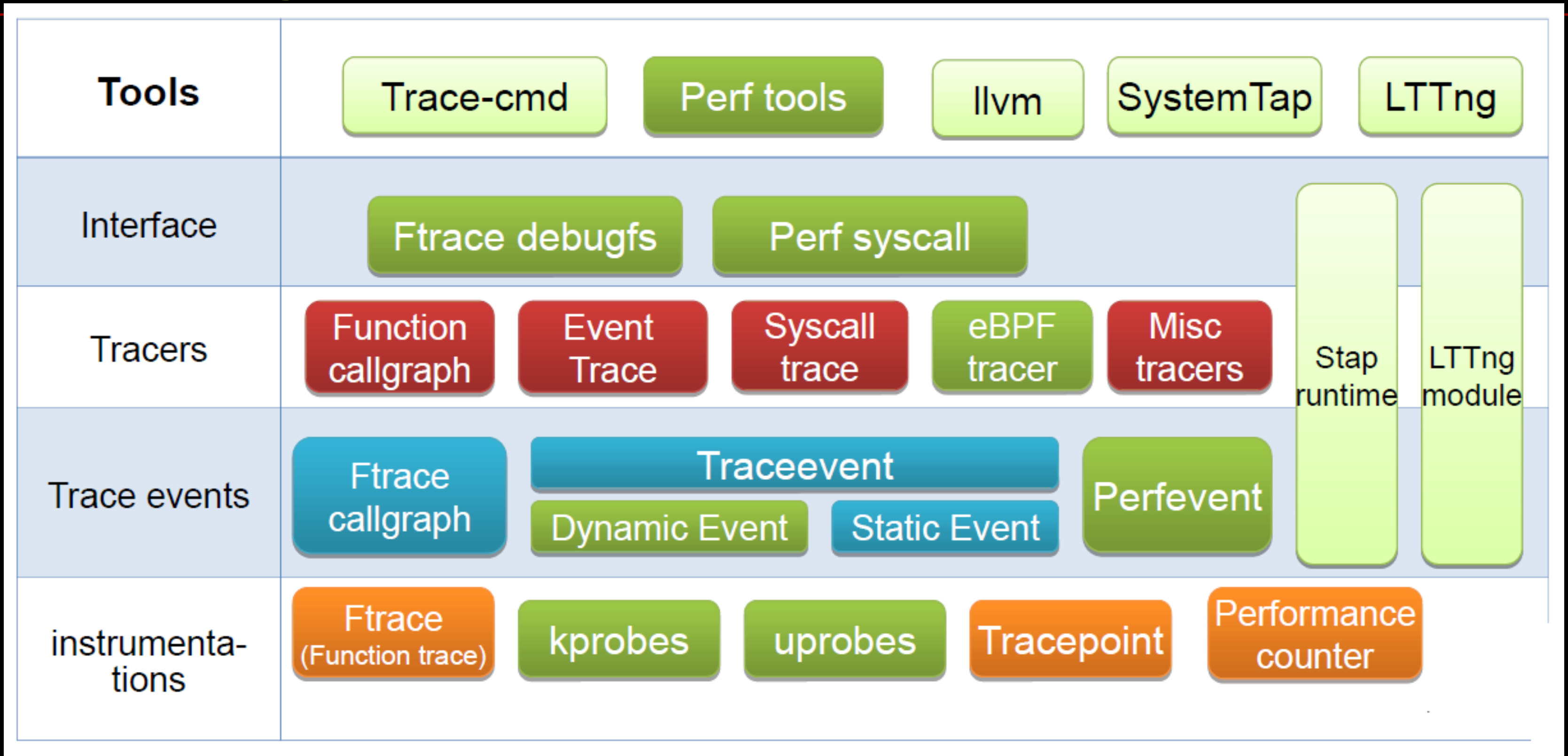
III. Python-assisted Data Center Dev
- HPC
- Software-Defined Everything

IV. Summary

I. eBPF & BCC

Background

- The Tracing Landscape



Tools	Trace-cmd	Perf tools	llvm	SystemTap	LTTng		
Interface	Ftrace debugfs	Perf syscall				Stap runtime	LTTng module
Tracers	Function callgraph	Event Trace	Syscall trace	eBPF tracer	Misc tracers		
Trace events	Ftrace callgraph	Traceevent Dynamic Event Static Event	Perfevent				
instrumentations	Ftrace (Function trace)	kprobes	uprobes	Tracepoint	Performance counter		

Source: http://tracingsummit.org/w/images/8/8c/TracingSummit2015-DynamicProbes.pdf

1) eBPF

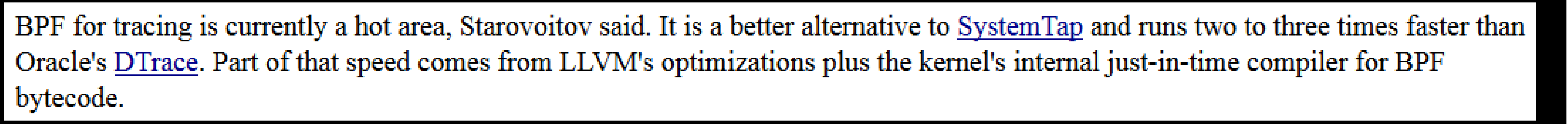
- https://en.wikipedia.org/wiki/Berkeley_Packet_Filter
- http://lwn.net/

BPF (Berkeley Packet Filter, aka cBPF)

- Introduced in kernel 2.1.75 (1997)
- Originally designed for packet filtering (tcpdump…)
- Apply for seccomp filters, traffic control…

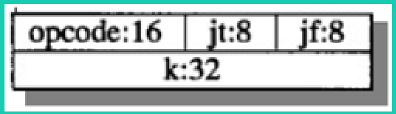
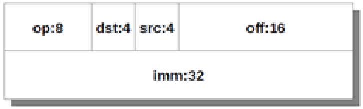
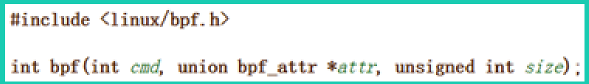
eBPF (extended BPF)

- Since Linux Kernel v3.15 and ongoing
- Aims at being a universal in-kernel virtual machine, it changes the old ways for Kernel instrumentation
- https://lwn.net/Articles/655544/



BPF for tracing is currently a hot area, Starovoitov said. It is a better alternative to SystemTap and runs two to three times faster than Oracle's DTrace. Part of that speed comes from LLVM's optimizations plus the kernel's internal just-in-time compiler for BPF bytecode.

Comparison

	cBPF	eBPF
Register	**Two 32 bit registers:** A: accumulator X: indexing	**Eleven 64 bit registers:** R0: return value/exit value R1-R5: arguments R6-R9: callee saved registers R10: read-only frame pointer
Instruction	~30  opcode:16 | jt:8 | jf:8 k:32	~90  op:8 | dst:4 | src:4 | off:16 imm:32
JIT	Support	**Support** (better mapping with newer architectures for JITing)
Toolchain	GCC, tools/net	LLVM eBPF backend
Platform	x86_64, ARM, ARM64, SPARC, PowerPC, MIPS and s390	x86-64, aarch64, s390x
System Call		 `#include <linux/bpf.h>` `int bpf(int cmd, union bpf_attr *attr, unsigned int size);` (CALL, MAP, LOAD...)

Dev

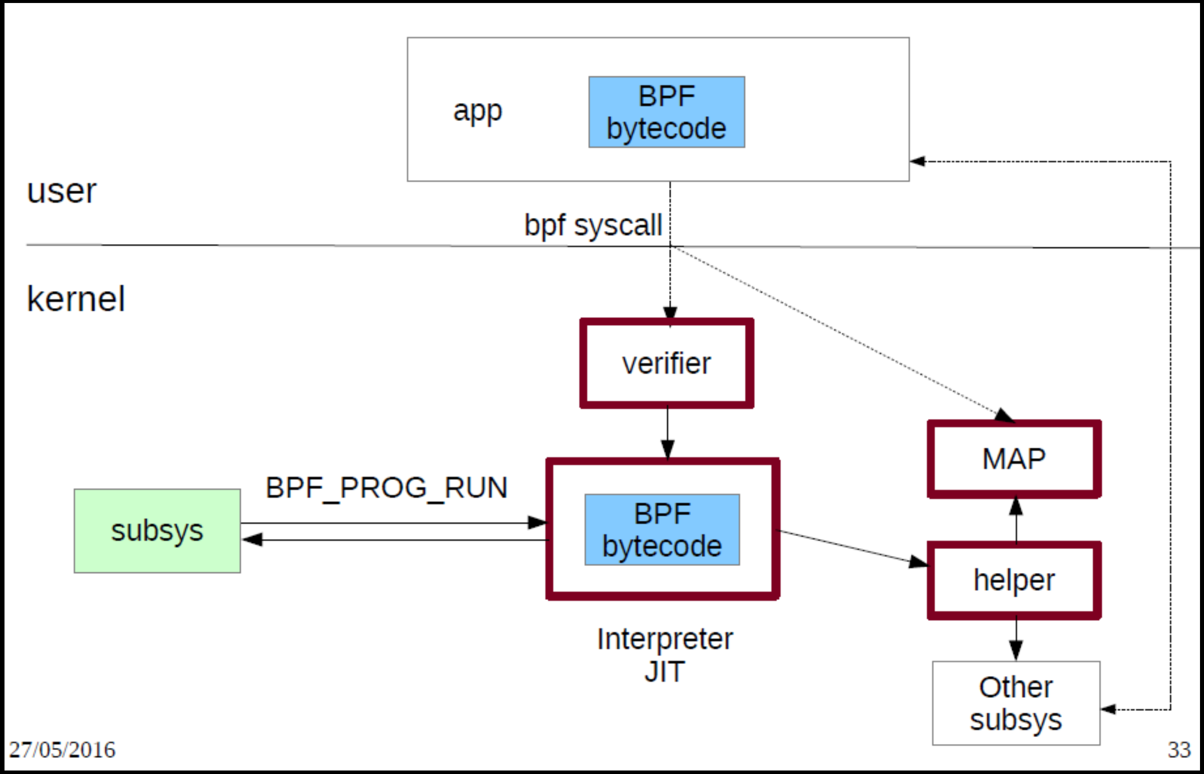
Dev Methods

1) write directly using eBPF assembly

2) Write it using C, and compile with LLVM

3) BCC

…



Source: http://www.slideshare.net/vh21/meet-cutebetweenebpfandtracing

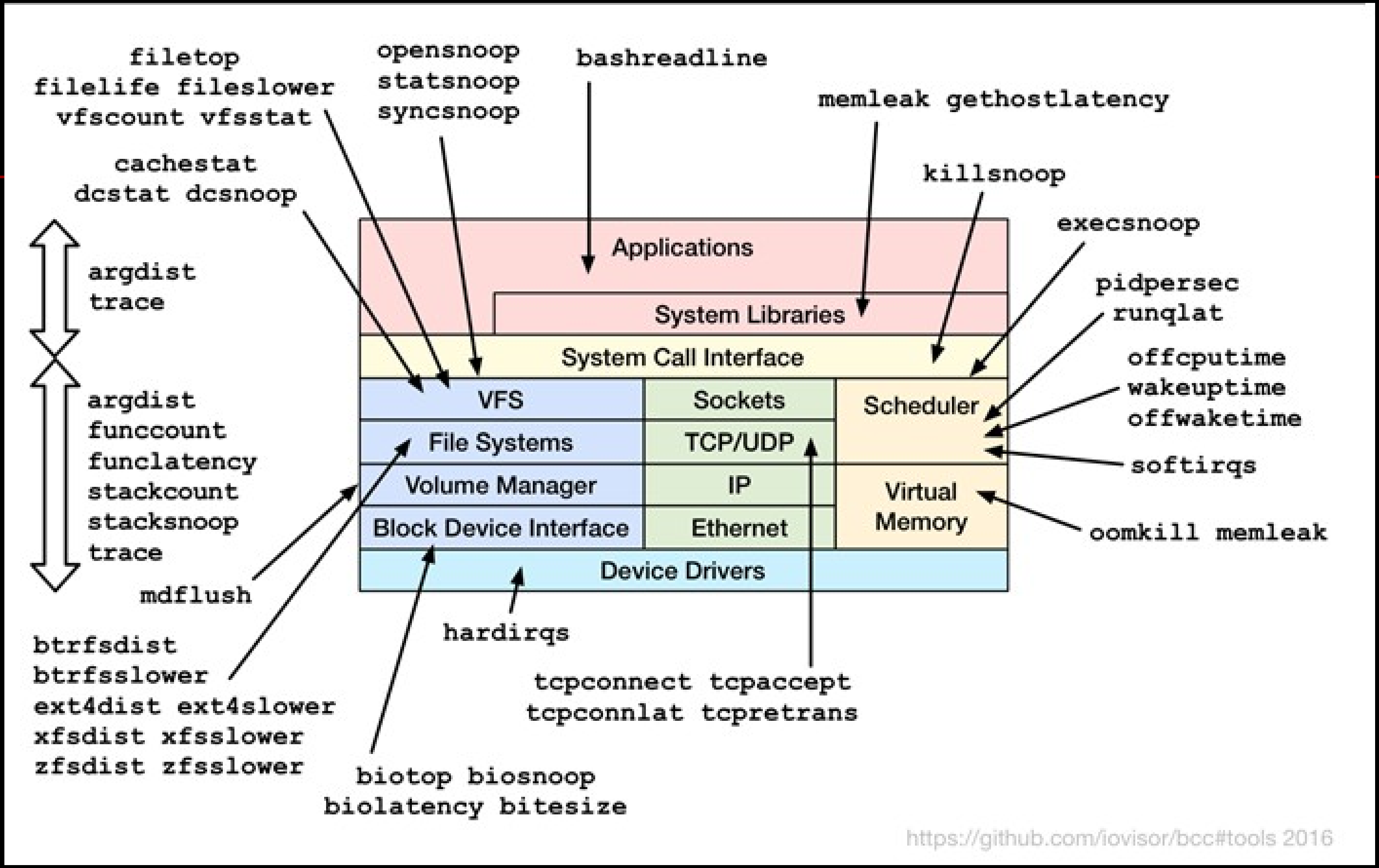
2) BCC (BPF Compiler Collection)

- https://iovisor.github.io/bcc/
- https://github.com/iovisor/bcc.git

A toolkit with Python/Lua frontend for compiling, loading, and executing BPF programs, which allows user-defined instrumentation on a live kernel image:

- Compile BPF program from C source
- Attach BPF program to kprobe/uprobe/tracepoint/USDT/socket
- Poll data from BPF program
- Framework for building new tools or one-off scripts
- …

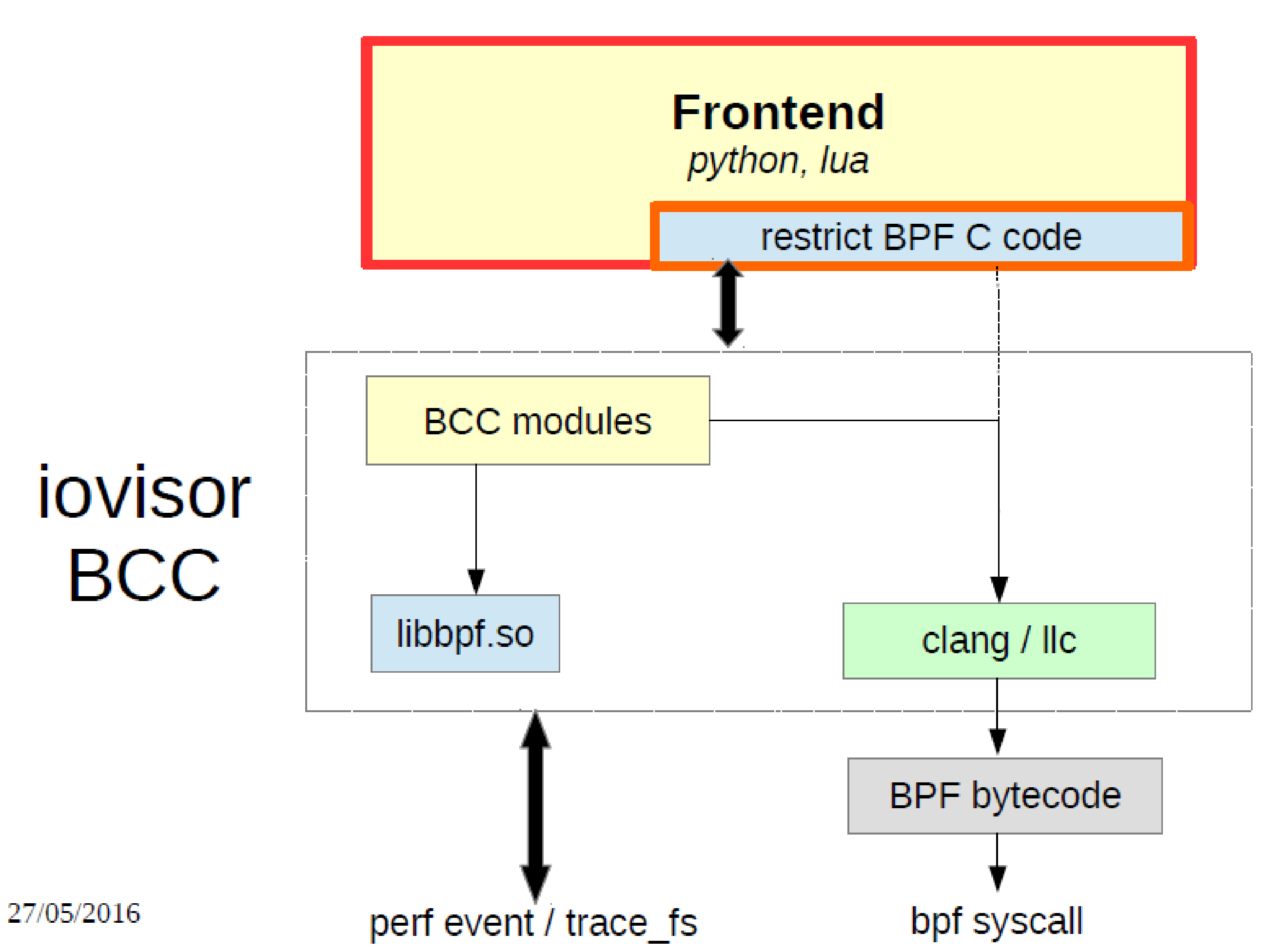
- BCC tools



https://github.com/iovisor/bcc#tools 2016

- Python is given the fullest support

Arch



Frontend

python, lua

restrict BPF C code

iovisor BCC

BCC modules

libbpf.so

clang / llc

BPF bytecode

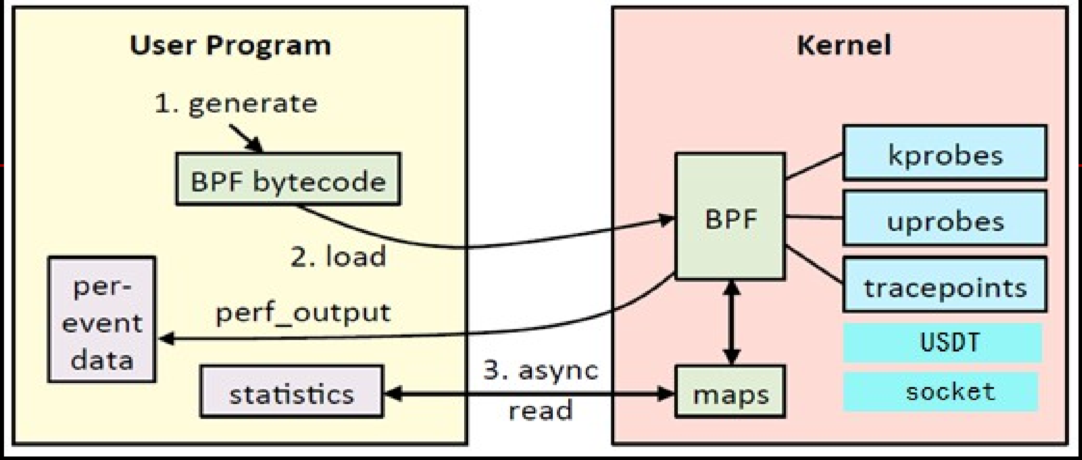
perf event / trace_fs

bpf syscall

27/05/2016

Source: http://www.slideshare.net/vh21/meet-cutebetweenebpfandtracing

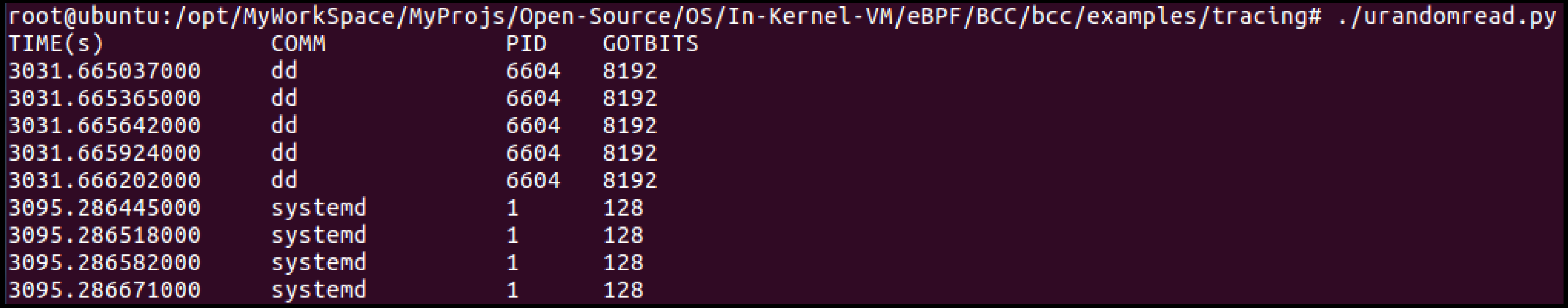
For Tracing



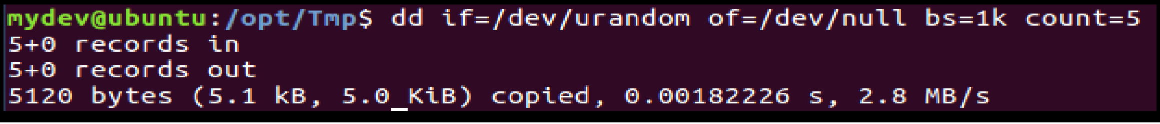
Source: http://www.slideshare.net/brendangregg/linux-bpf-superpowers

Sample

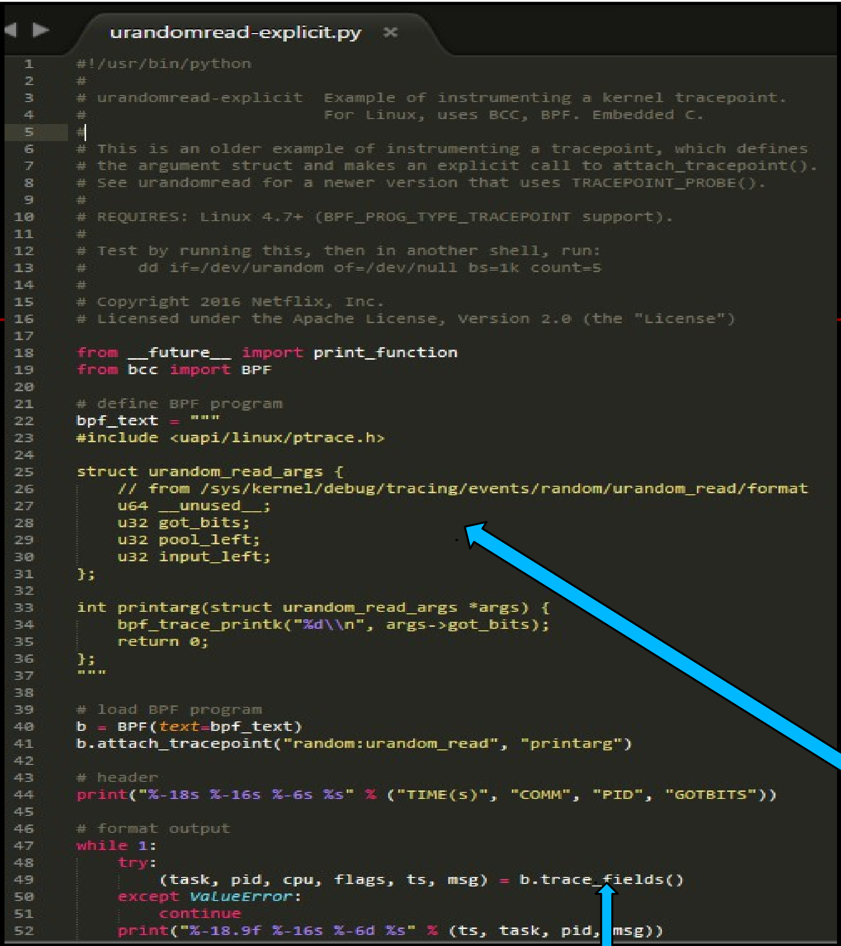
- bcc/examples/tracing/urandomread*.*



```
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/eBPF/BCC/bcc/examples/tracing# ./urandomread.py
TIME(s)            COMM             PID    GOTBITS
3031.665037000     dd               6604   8192
3031.665365000     dd               6604   8192
3031.665642000     dd               6604   8192
3031.665924000     dd               6604   8192
3031.666202000     dd               6604   8192
3095.286445000     systemd          1      128
3095.286518000     systemd          1      128
3095.286582000     systemd          1      128
3095.286671000     systemd          1      128
```

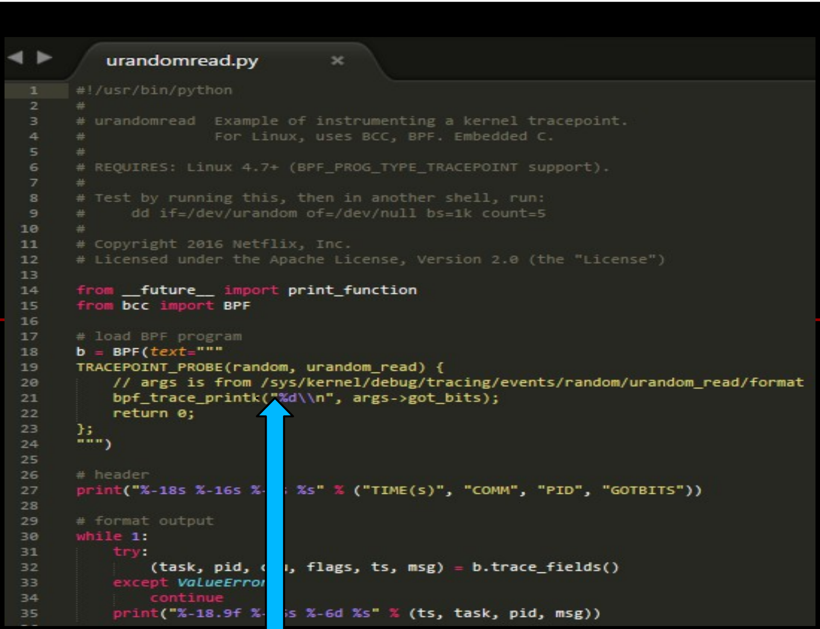


```
mydev@ubuntu:/opt/Tmp$ dd if=/dev/urandom of=/dev/null bs=1k count=5
5+0 records in
5+0 records out
5120 bytes (5.1 kB, 5.0 KiB) copied, 0.00182226 s, 2.8 MB/s
```



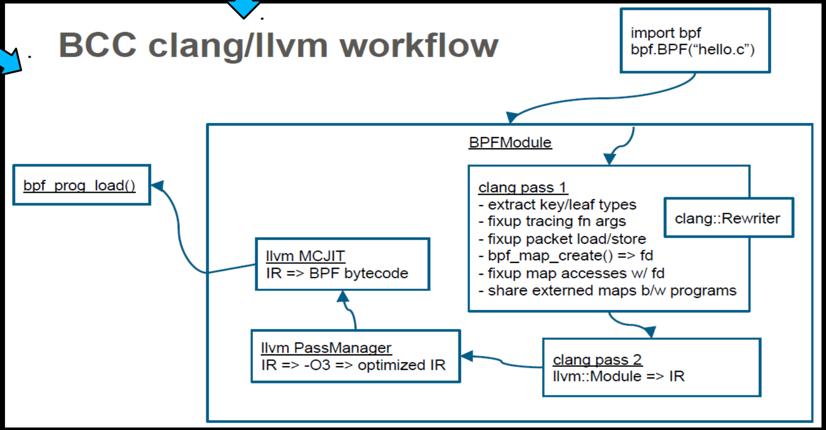
urandomread-explicit.py

```python
#!/usr/bin/python
#
# urandomread-explicit  Example of instrumenting a kernel tracepoint.
#                       For Linux, uses BCC, BPF. Embedded C.
#
# This is an older example of instrumenting a tracepoint, which defines
# the argument struct and makes an explicit call to attach_tracepoint().
# See urandomread for a newer version that uses TRACEPOINT_PROBE().
#
# REQUIRES: Linux 4.7+ (BPF_PROG_TYPE_TRACEPOINT support).
#
# Test by running this, then in another shell, run:
#     dd if=/dev/urandom of=/dev/null bs=1k count=5
#
# Copyright 2016 Netflix, Inc.
# Licensed under the Apache License, Version 2.0 (the "License")

from __future__ import print_function
from bcc import BPF

# define BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>

struct urandom_read_args {
    // from /sys/kernel/debug/tracing/events/random/urandom_read/format
    u64 __unused__;
    u32 got_bits;
    u32 pool_left;
    u32 input_left;
};

int printarg(struct urandom_read_args *args) {
    bpf_trace_printk("%d\\n", args->got_bits);
    return 0;
};
"""

# load BPF program
b = BPF(text=bpf_text)
b.attach_tracepoint("random:urandom_read", "printarg")

# header
print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "GOTBITS"))

# format output
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        continue
    print("%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```

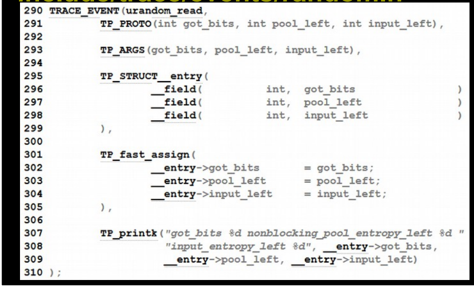


urandomread.py

```python
#!/usr/bin/python
#
# urandomread  Example of instrumenting a kernel tracepoint.
#              For Linux, uses BCC, BPF. Embedded C.
#
# REQUIRES: Linux 4.7+ (BPF_PROG_TYPE_TRACEPOINT support).
#
# Test by running this, then in another shell, run:
#     dd if=/dev/urandom of=/dev/null bs=1k count=5
#
# Copyright 2016 Netflix, Inc.
# Licensed under the Apache License, Version 2.0 (the "License")

from __future__ import print_function
from bcc import BPF

# load BPF program
b = BPF(text="""
TRACEPOINT_PROBE(random, urandom_read) {
    // args is from /sys/kernel/debug/tracing/events/random/urandom_read/format
    bpf_trace_printk("%d\\n", args->got_bits);
    return 0;
};
""")

# header
print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "GOTBITS"))

# format output
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        continue
    print("%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```



include/trace/events/random.h



```c
290 TRACE_EVENT(urandom_read,
291         TP_PROTO(int got_bits, int pool_left, int input_left),
292 
293         TP_ARGS(got_bits, pool_left, input_left),
294 
295         TP_STRUCT__entry(
296                 __field(          int,  got_bits                 )
297                 __field(          int,  pool_left                )
298                 __field(          int,  input_left               )
299         ),
300 
301         TP_fast_assign(
302                 __entry->got_bits       = got_bits;
303                 __entry->pool_left      = pool_left;
304                 __entry->input_left     = input_left;
305         ),
306 
307         TP_printk("got_bits %d nonblocking_pool_entropy_left %d "
308                 "input_entropy_left %d", __entry->got_bits,
309                 __entry->pool_left, __entry->input_left)
310 );
```

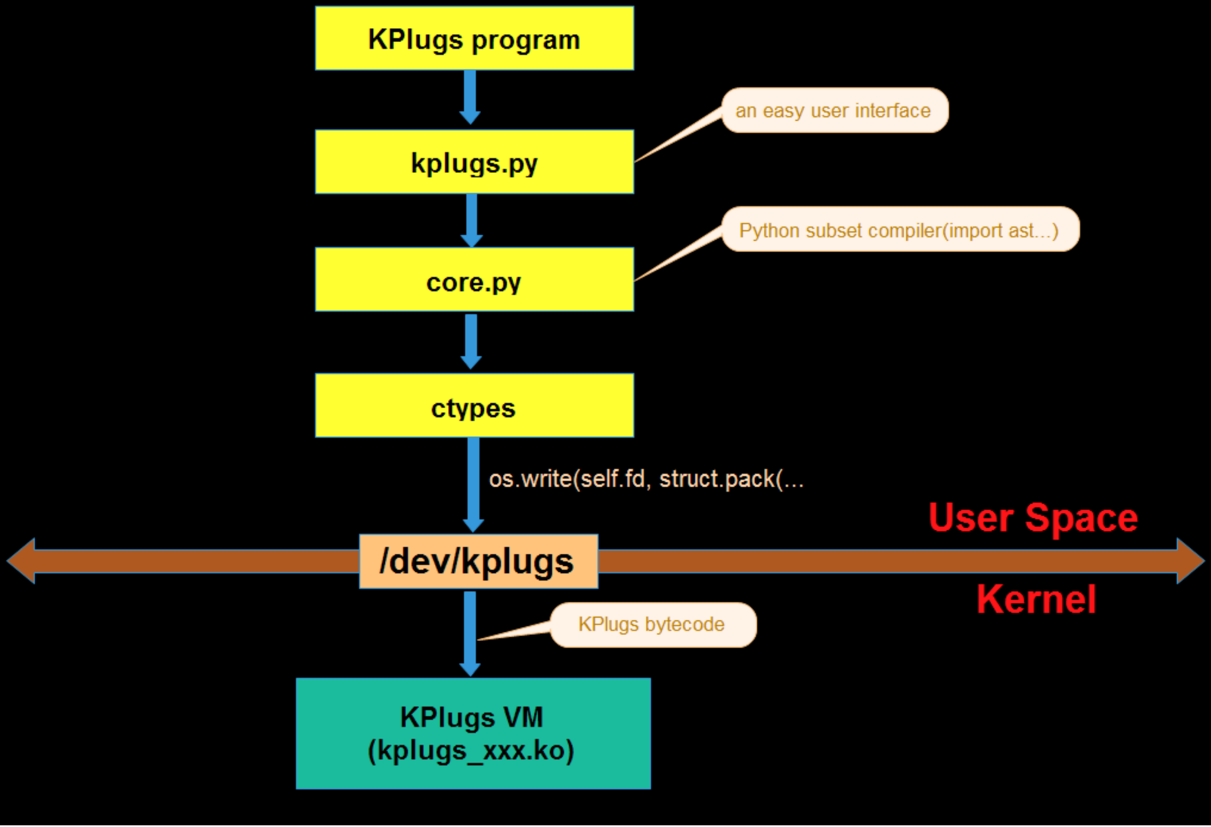
/sys/kernel/debug/tracing/trace_pipe

Source: http://linuxplumbersconf.org/2015/ocw//system/presentations/3249/original/bpf_llvm_2015aug19.pdf

II. In-Kernel Virtual Machine

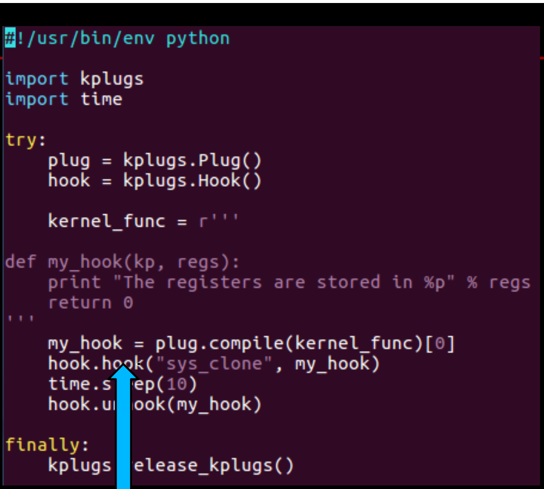
1) KPlugs

- https://www.kplugs.org
- https://github.com/avielw/kplugs
- Arch

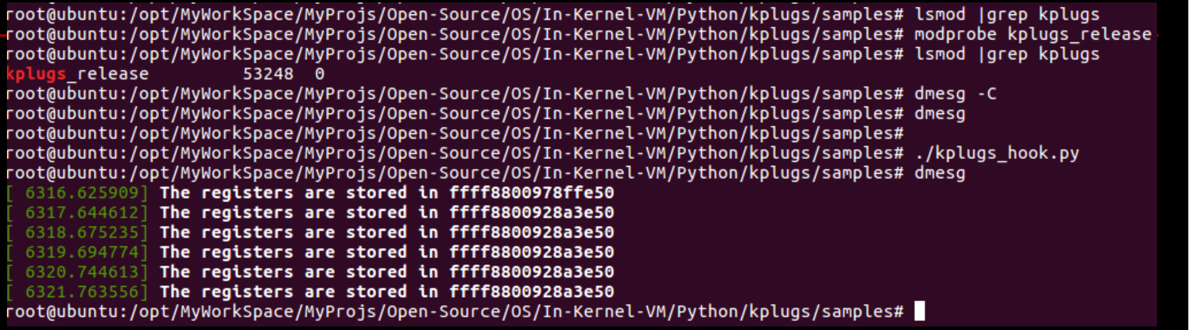


Hook sample

- kplugs.Mem - access all of the computer's memory of the computer (kernel and all processes' user space)
- kplugs.Symbol - resolve kernel symbols
- kplugs.Hook - hook kernel functions with a KPlugs function
- kplugs.Caller - call an exported kernel function

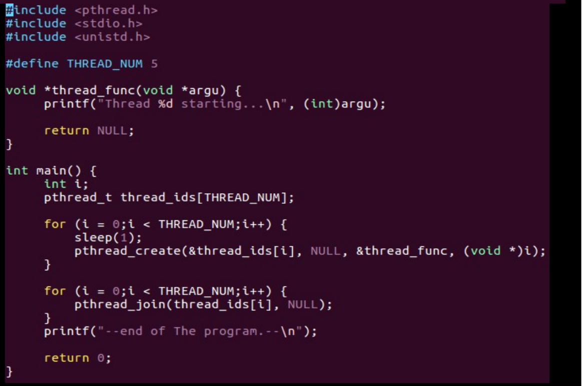


```python
#!/usr/bin/env python

import kplugs
import time

try:
    plug = kplugs.Plug()
    hook = kplugs.Hook()

    kernel_func = r'''

def my_hook(kp, regs):
    print "The registers are stored in %p" % regs
    return 0
'''
    my_hook = plug.compile(kernel_func)[0]
    hook.hook("sys_clone", my_hook)
    time.sleep(10)
    hook.unhook(my_hook)

finally:
    kplugs.release_kplugs()
```



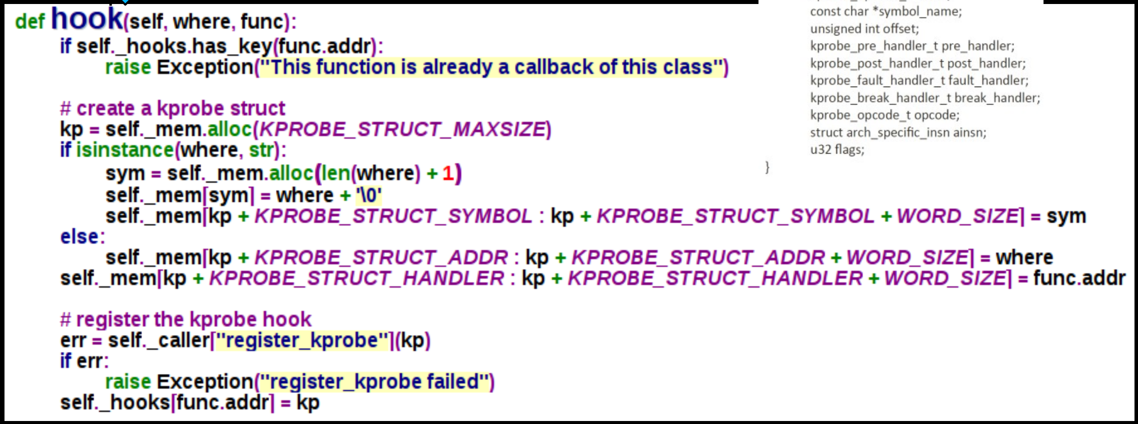
```
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# lsmod |grep kplugs
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# modprobe kplugs_release
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# lsmod |grep kplugs
kplugs_release         53248  0
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# dmesg -C
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# dmesg
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples#
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# ./kplugs_hook.py
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples# dmesg
[ 6316.625909] The registers are stored in ffff8800978ffe50
[ 6317.644612] The registers are stored in ffff8800928a3e50
[ 6318.675235] The registers are stored in ffff8800928a3e50
[ 6319.694774] The registers are stored in ffff8800928a3e50
[ 6320.744613] The registers are stored in ffff8800928a3e50
[ 6321.763556] The registers are stored in ffff8800928a3e50
root@ubuntu:/opt/MyWorkSpace/MyProjs/Open-Source/OS/In-Kernel-VM/Python/kplugs/samples#
```

```
mydev@ubuntu:/opt/MyWorkSpace/Test/Linux/Thread$ ./multithread
```



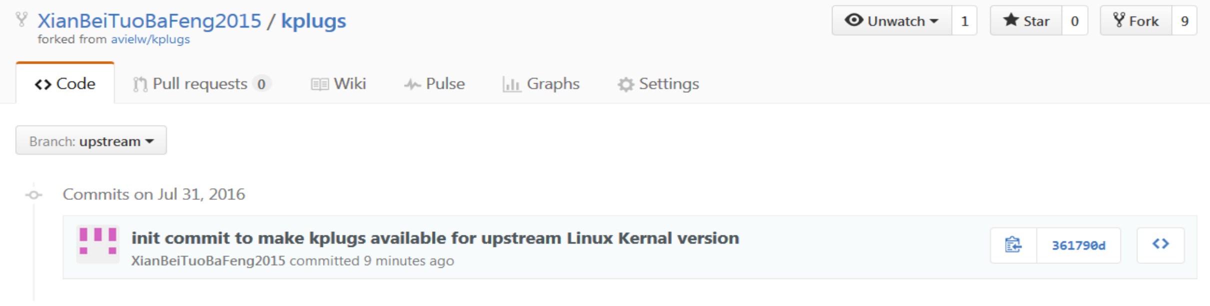
```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define THREAD_NUM 5

void *thread_func(void *argu) {
     printf("Thread %d starting...\n", (int)argu);

     return NULL;
}

int main() {
     int i;
     pthread_t thread_ids[THREAD_NUM];

     for (i = 0;i < THREAD_NUM;i++) {
         sleep(1);
         pthread_create(&thread_ids[i], NULL, &thread_func, (void *)i);
     }

     for (i = 0;i < THREAD_NUM;i++) {
         pthread_join(thread_ids[i], NULL);
     }
     printf("--end of The program.--\n");

     return 0;
}
```

```c
struct kprobe {
        struct hlist_node hlist;
        struct lit_head list;
        unsigned long nmissed;
        kprobe_opcode_t *addr;
        const char *symbol_name;
        unsigned int offset;
        kprobe_pre_handler_t pre_handler;
        kprobe_post_handler_t post_handler;
        kprobe_fault_handler_t fault_handler;
        kprobe_break_handler_t break_handler;
        kprobe_opcode_t opcode;
        struct arch_specific_insn ainsn;
        u32 flags;
}
```



```python
def hook(self, where, func):
    if self._hooks.has_key(func.addr):
        raise Exception("This function is already a callback of this class")

    # create a kprobe struct
    kp = self._mem.alloc(KPROBE_STRUCT_MAXSIZE)
    if isinstance(where, str):
        sym = self._mem.alloc(len(where) + 1)
        self._mem[sym] = where + '\0'
        self._mem[kp + KPROBE_STRUCT_SYMBOL : kp + KPROBE_STRUCT_SYMBOL + WORD_SIZE] = sym
    else:
        self._mem[kp + KPROBE_STRUCT_ADDR : kp + KPROBE_STRUCT_ADDR + WORD_SIZE] = where
    self._mem[kp + KPROBE_STRUCT_HANDLER : kp + KPROBE_STRUCT_HANDLER + WORD_SIZE] = func.addr

    # register the kprobe hook
    err = self._caller["register_kprobe"](kp)
    if err:
        raise Exception("register_kprobe failed")
    self._hooks[func.addr] = kp
```

- **My Fork**
  https://github.com/XianBeiTuoBaFeng2015/kplugs



XianBeiTuoBaFeng2015 / kplugs
forked from aviewl/kplugs

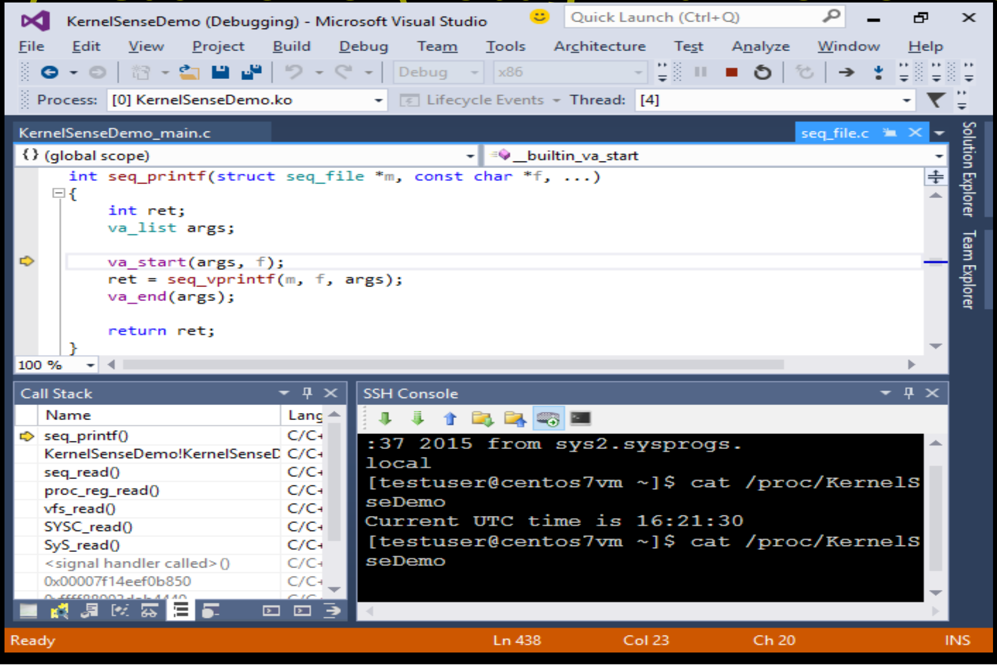
Branch: upstream

Commits on Jul 31, 2016

init commit to make kplugs available for upstream Linux Kernal version
XianBeiTuoBaFeng2015 committed 9 minutes ago

361790d

- **Debugging**
  1) python -m trace --trace ./kplugs_hello.py
  2) VisualKernel (Debug Linux Kernel with Visual Studio)



KernelSenseDemo (Debugging) - Microsoft Visual Studio

Process: [0] KernelSenseDemo.ko
Thread: [4]

(global scope)
__builtin_va_start

```
int seq_printf(struct seq_file *m, const char *f, ...)
{
    int ret;
    va_list args;

    va_start(args, f);
    ret = seq_vprintf(m, f, args);
    va_end(args);

    return ret;
}
```

Call Stack

Name	Lang
seq_printf()	C/C+
KernelSenseDemo!KernelSenseD	C/C+
seq_read()	C/C+
proc_reg_read()	C/C+
vfs_read()	C/C+
SYSC_read()	C/C+
SyS_read()	C/C+
<signal handler called> ()	C/C+
0x00007f14eef0b850	C/C+

SSH Console

```
:37 2015 from sys2.sysprogs.
local
[testuser@centos7vm ~]$ cat /proc/KernelS
seDemo
Current UTC time is 16:21:30
[testuser@centos7vm ~]$ cat /proc/KernelS
seDemo
```

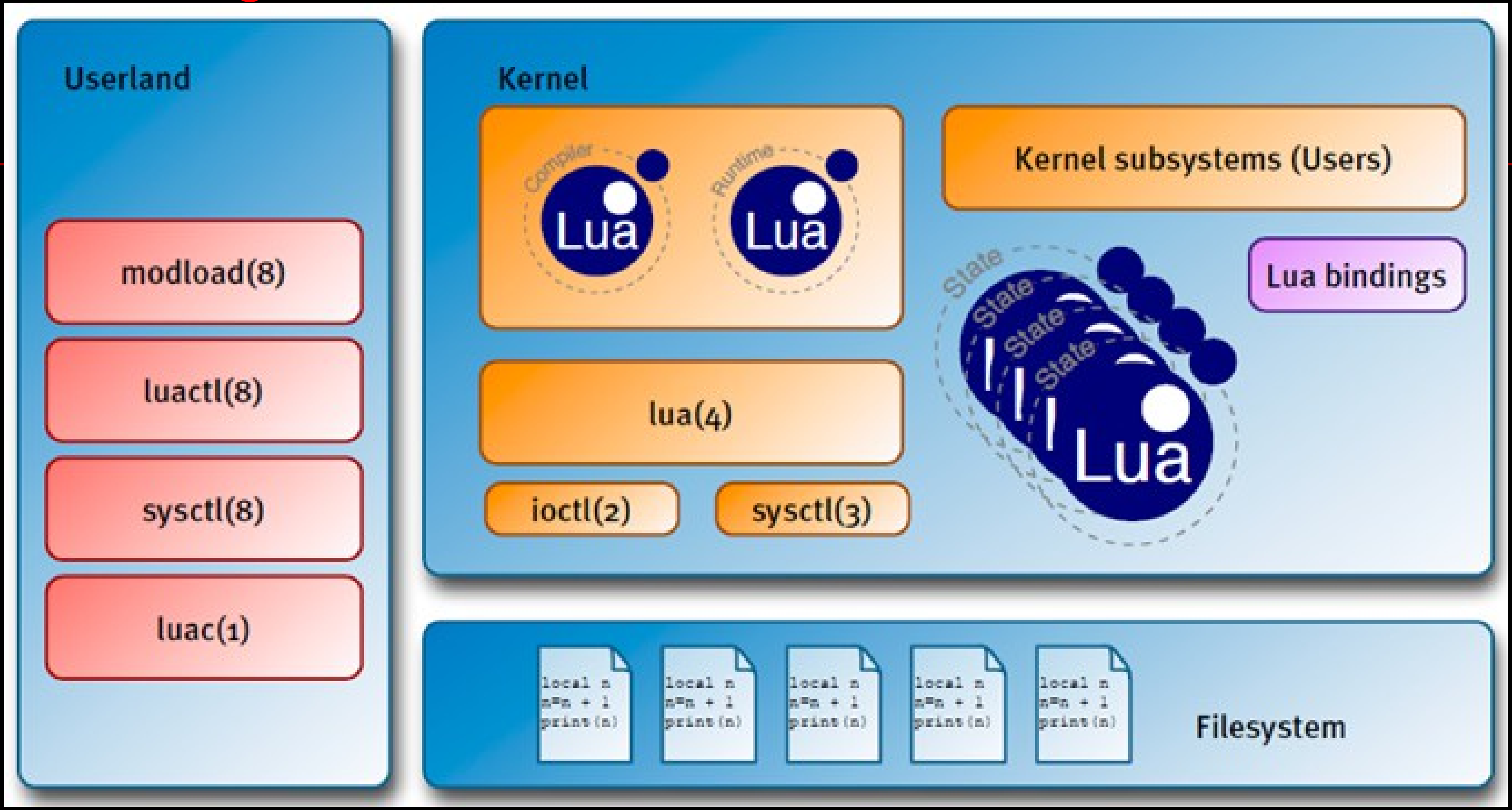
Ready Ln 438 Col 23 Ch 20 INS

2) Application

- http://www.netbsd.org
- http://www.lua.org
- NetBSD Kernel scripting with Lua

- be part of NetBSD 6 (Userland)
- be part of NetBSD 7 (Kernel)

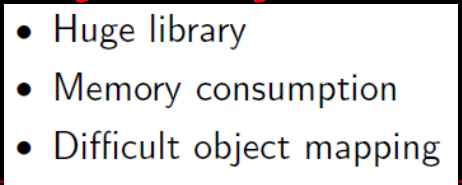
- The Big Picture



Source: https://archive.fosdem.org/2013/schedule/event/lua_in_the_netbsd_kernel/

- still lack of JIT(Just-in-Time) support, but available on Linux

Why not Python?

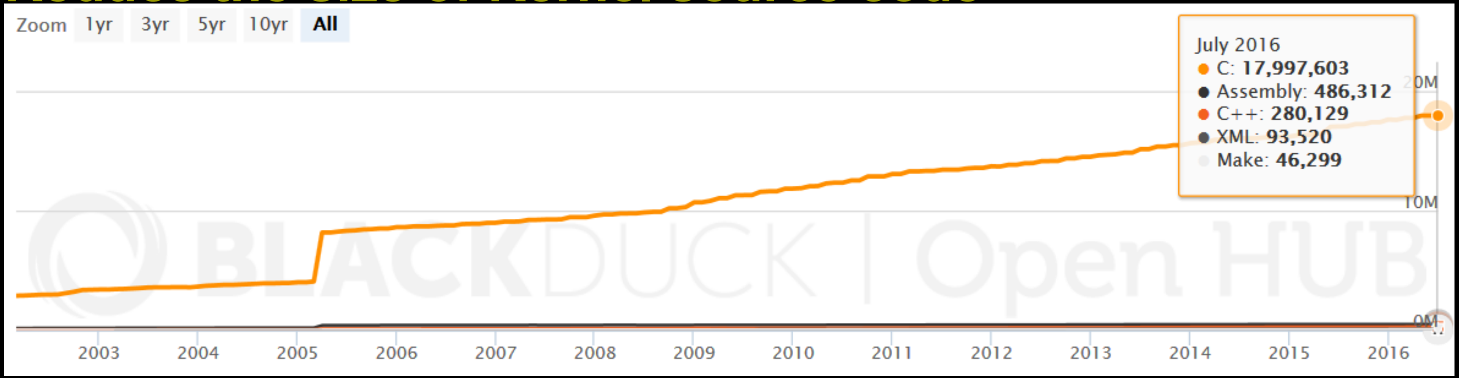


- Huge library
- Memory consumption
- Difficult object mapping

VM size at megabytes level, but the key idea behind KPlugs is…

Conclusions

Reduce the size of Kernel source code



Source: https://www.openhub.net/p/linux/analyses/latest/languages_summary

Deliver a higher-level programming environment to the Kernel

Let users explore the system in an easy way

Great innovation in OS development!

III. Python-assisted Data Center Dev

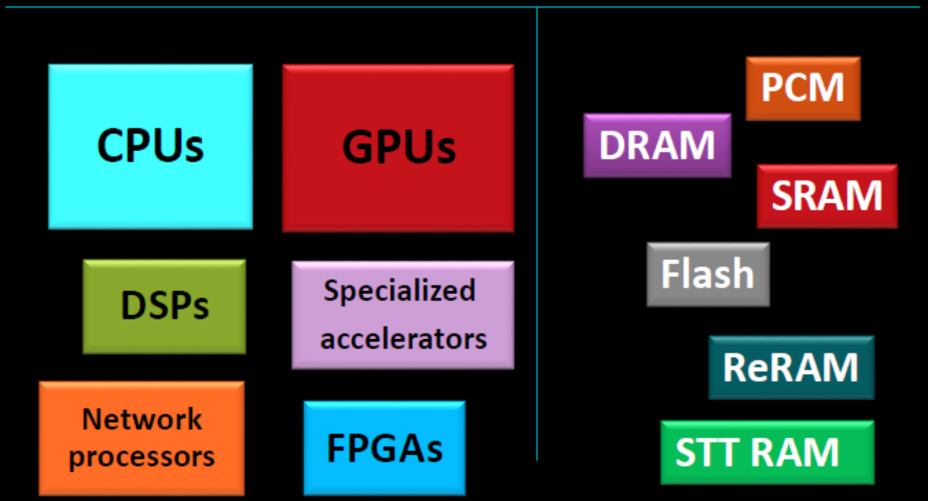
1) HPC

- High Performance Computing or Heterogeneous Parallel Computing

C/C++ FORTRAN Java

UPC/UPC++ python MPI

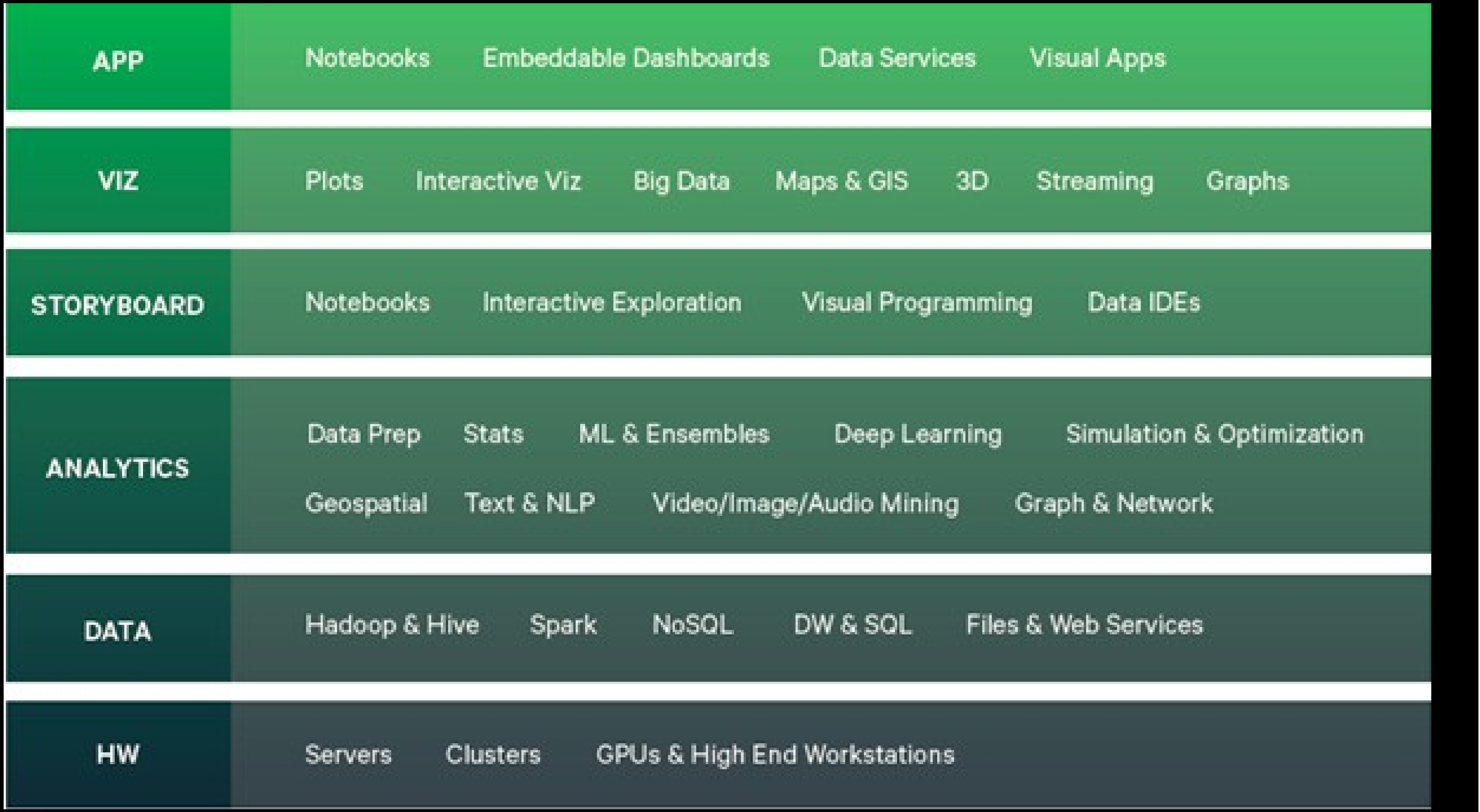
Kokkos/RAJA OpenMP OpenACC



Heterogeneity Options

Anaconda

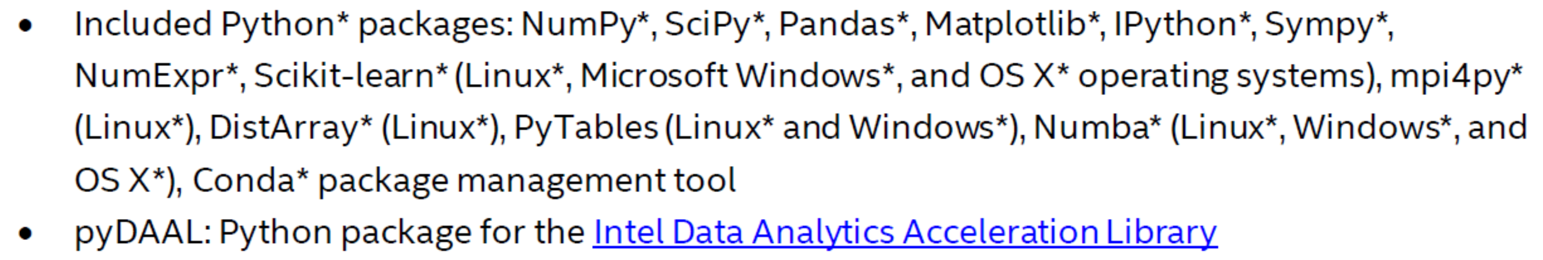
- https://www.continuum.io/
- Open Source Modern Analytics Platform Powered by Python
- Swissknife for Big Data, AI, HPC, Cloud/Web, Exploration & Viz



APP	Notebooks Embeddable Dashboards Data Services Visual Apps
VIZ	Plots Interactive Viz Big Data Maps & GIS 3D Streaming Graphs
STORYBOARD	Notebooks Interactive Exploration Visual Programming Data IDEs
ANALYTICS	Data Prep Stats ML & Ensembles Deep Learning Simulation & Optimization Geospatial Text & NLP Video/Image/Audio Mining Graph & Network
DATA	Hadoop & Hive Spark NoSQL DW & SQL Files & Web Services
HW	Servers Clusters GPUs & High End Workstations

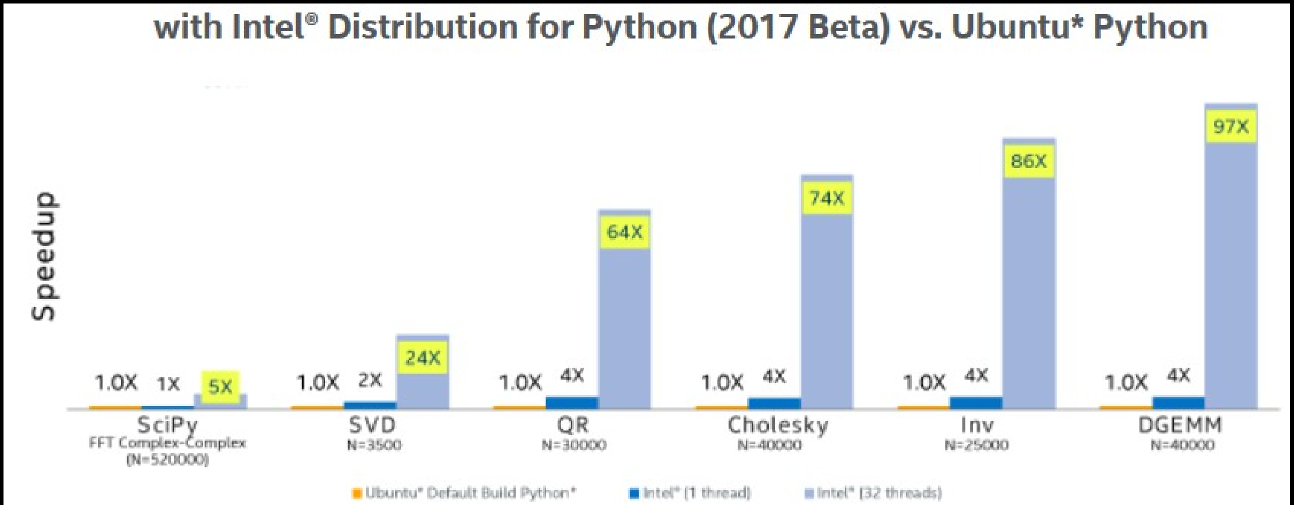
Intel

- Intel Distribution for Python
- https://software.intel.com/en-us/python-distribution
- accelerated with MKL, MPI, TBB, DAAL
- support of Xeon Phi
- packages:



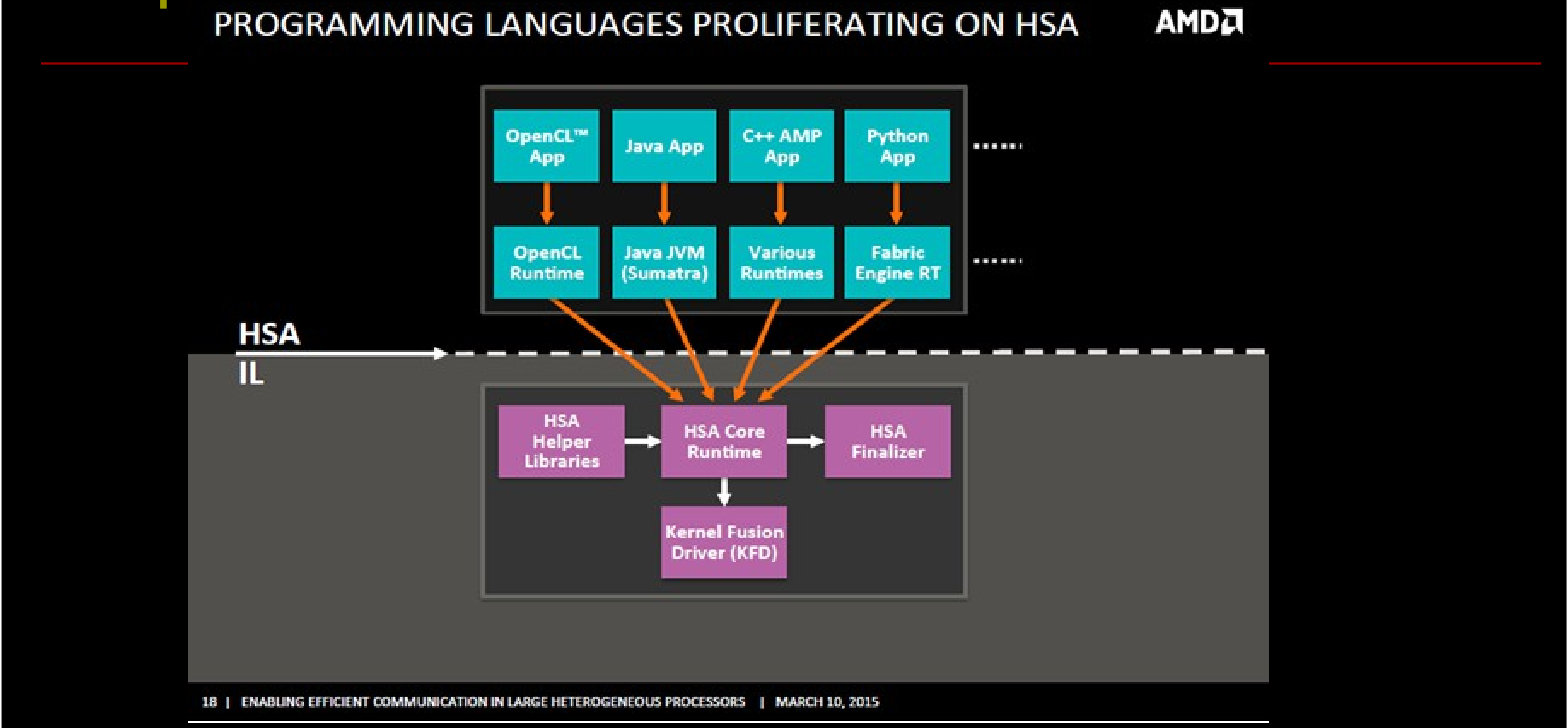
- Included Python* packages: NumPy*, SciPy*, Pandas*, Matplotlib*, IPython*, Sympy*, NumExpr*, Scikit-learn* (Linux*, Microsoft Windows*, and OS X* operating systems), mpi4py* (Linux*), DistArray* (Linux*), PyTables (Linux* and Windows*), Numba* (Linux*, Windows*, and OS X*), Conda* package management tool
- pyDAAL: Python package for the Intel Data Analytics Acceleration Library

- Performance Boost on Select Numerical Functions



AMD

- Heterogeneous System Architecture
- http://www.hsafoundation.com/



PROGRAMMING LANGUAGES PROLIFERATING ON HSA

AMD

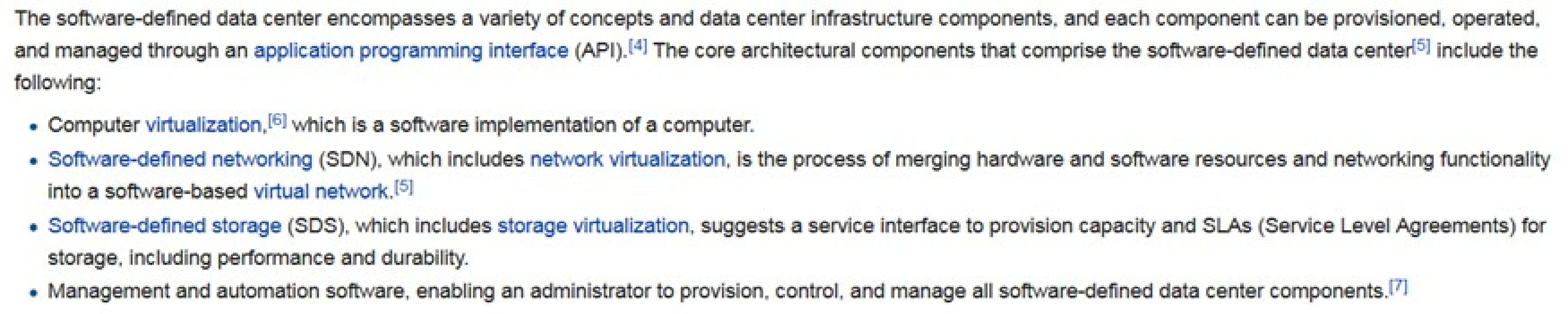
18 | ENABLING EFFICIENT COMMUNICATION IN LARGE HETEROGENEOUS PROCESSORS | MARCH 10, 2015

- https://github.com/ContinuumIO/Numba-HSA-Webinar/

2) Software-Defined Everything (SDE)

- Software Defined Data Center (SDDC)
- Software-Defined Networking (SDN)
- Software-Defined Storage (SDS)

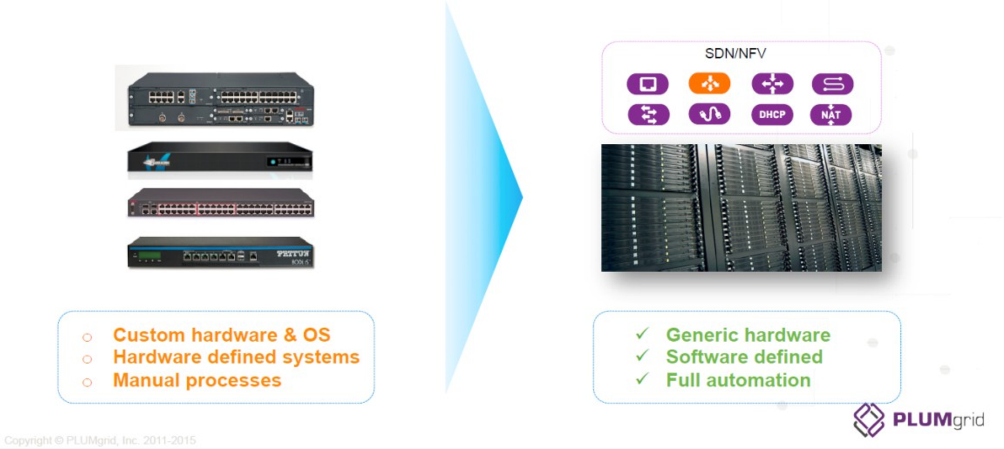
- https://en.wikipedia.org/wiki/Software-defined_data_center



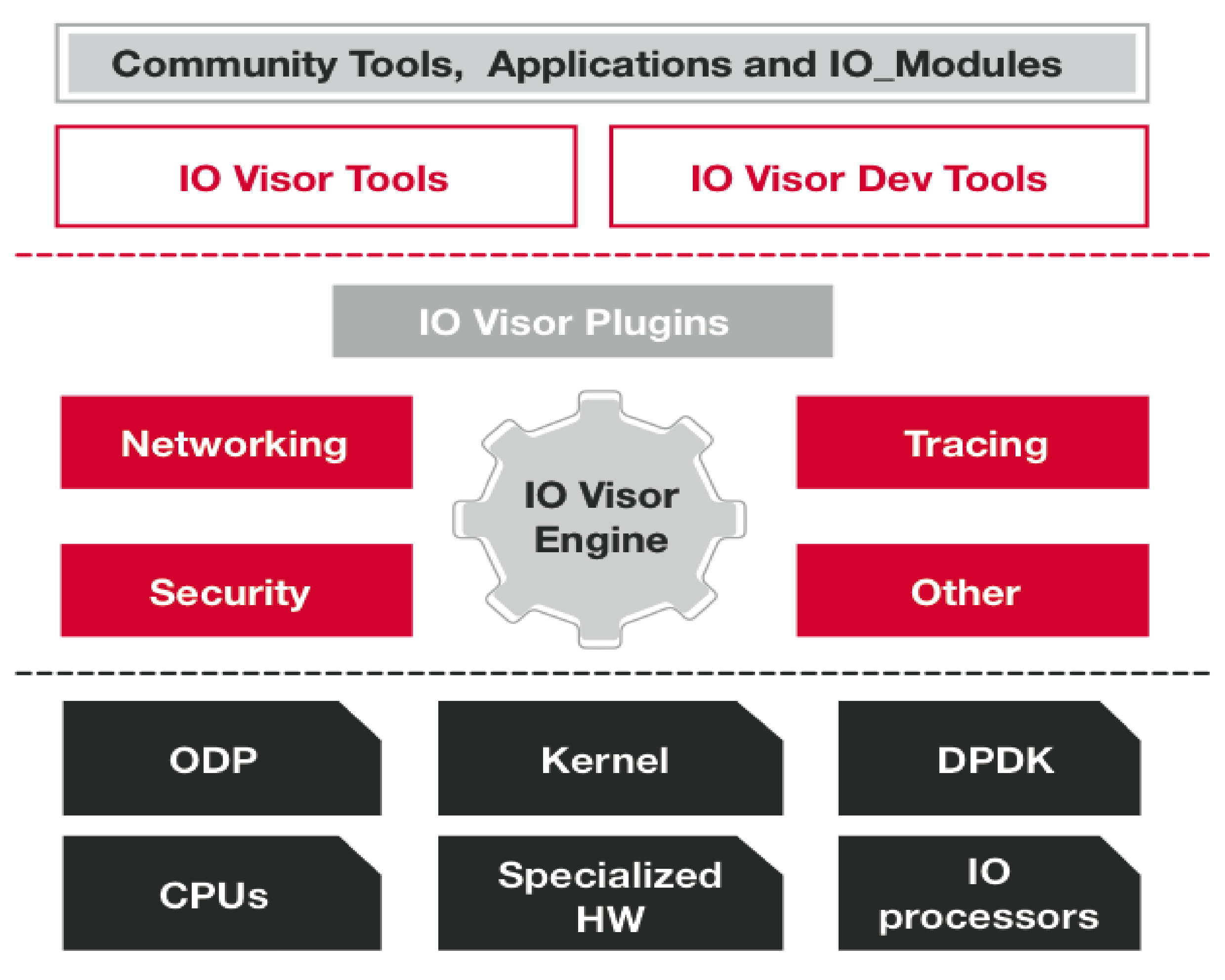
The software-defined data center encompasses a variety of concepts and data center infrastructure components, and each component can be provisioned, operated, and managed through an application programming interface (API).[4] The core architectural components that comprise the software-defined data center[5] include the following:

- Computer virtualization,[6] which is a software implementation of a computer.
- Software-defined networking (SDN), which includes network virtualization, is the process of merging hardware and software resources and networking functionality into a software-based virtual network.[5]
- Software-defined storage (SDS), which includes storage virtualization, suggests a service interface to provision capacity and SLAs (Service Level Agreements) for storage, including performance and durability.
- Management and automation software, enabling an administrator to provision, control, and manage all software-defined data center components.[7]

- Traditional to Software Defined Systems



- https://www.iovisor.org/



https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4/

IV. Summary

- User space/Kernel space Repartition & Unifying
  - write your function in user space, while run it in kernel
  - user space drivers
  - scripting your OS
- Compilation technologies are rapidly evolving
  - compiler infrastructure like LLVM
  - source-to-source, bytecode-to-bytecode, and bytecode to native code compiler
- Python-based Domain Specific Languages & language subset

- Python is sure to play an important role in next generation Software Defined Systems!

Q & A

Thanks!

Reference

Slides/materials from many and varied sources:

- http://en.wikipedia.org/wiki/
- http://www.slideshare.net/
- https://www.kernel.org/doc/Documentation/
- http://tracingsummit.org
- http://www.brendangregg.com/blog/
- https://www.python.org
- http://llvm.org
- https://en.wikipedia.org/wiki/Just-in-time_compilation
- http://sysprogs.com/VisualKernel/
- https://www.netbsd.org/gallery/presentations/
- https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)
- https://www.opennetworking.org/
- https://www.opnfv.org/
- http://pypy.org
- https://en.wikipedia.org/wiki/Runtime_system
- https://en.wikipedia.org/wiki/Intermediate_representation
- …