Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

利用性能计数器来提供硬件事件（如 CPU 周期、指令、缓存命中等）和软件事件（如上下文切换、页面错误等）等的详细视图。

Hardware	Hardware cache	Software	Kernel PMU	Tracepoint
cpu-cycles	L1-dcache-load-misses	context-switches	context-switches	内核插桩
instructions	L1-icache-load-misses	cpu-clock	cpu-clock	
branch-misses	L1-dcache-loads / stores	task-clock	task-clock	
cache-misses	branch-loads[-misses]	minor/major-faults	minor/major-faults	
	dTLB-loads[-misses]	page-faults	page-faults	
	dTLB-stores[-misses]	cpu-migrations		
	iTLB-load-misses			

Command

stat: event counting
record: profiling / static tracing
reoprt: reporting
top: profiling

Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

例子：

stack trace (-g) for

L1-dcache-load-misses events (-e)

every 500 event (-c 500) / freq (-F 49)

on all CPUs (-a) / CPU0 (-C 0)

-p pid / -t tid

to out.data(-o)

for 2s (-- sleep 2)

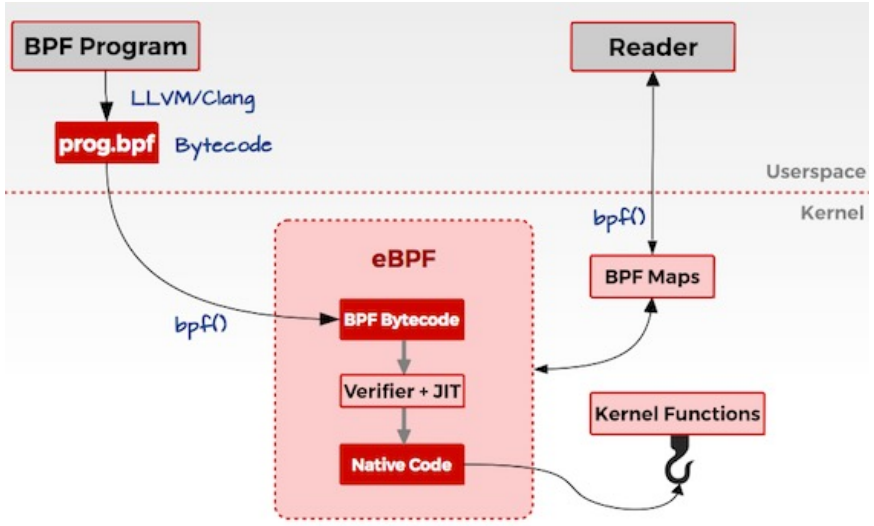
```
perf stat -d -p PID -- sleep 1
perf stat -e task-clock,cpu-clock -p PID -- sleep 1
perf stat -e sched:sched_switch -t TID -- sleep 1

perf record -F 499 -a -g -- sleep 5
perf record -e sched:sched_switch -t TID -- sleep 1

perf script -i out.data > out.perf
stackcollapse-perf.pl out.perf > out.folded
flamegraph.pl out.folded > out.svg

perf top -F 99 -a -g -p PID --no-children
```

Profiling 工具

perf　　eBPF（bcc、bpftrace）　Intel Processor Trace

BPF（Berkeley Packet Filter）一种运行内核层的网络数据包过滤和捕获机制；

eBPF 扩展了传统BPF的功能，可基于程序事件在内核直接高效安全执行特定代码的能力 。



字节码编译- 验证、加载 - 运行、写入缓冲 - 用户读取缓冲 、分析

- **kprobes**：内核中动态跟踪。内核维护，理论上可动态跟踪到所有符号在 /proc/kallsyms 但不在 /sys/kernel/debug/kprobes/blacklist 的
- **uprobes**：用户级别的动态跟踪。
- **tracepoints**：内核中静态跟踪。开发人员维护的跟踪点，能够提供稳定的 ABI 接口，但是需维护，数量和场景受限。（用户静态探针 USDT ）
- **perf_events**：定时采样和PMC。

```
struct bpf_insn {
__u8            code;           /* opcode */
__u8            dst_reg:4;      /* dest register */
__u8            src_reg:4;      /* source register */
__s16           off;            /* signed offset */
__s32           imm;            /* signed immediate constant */};
msb                                                 lsb
+--------------------+---------------+----+----+-------+|
immediate             |offset         |src |dst |opcode  |
+--------------------+---------------+----+----+-------+|
```

支持从通用内存（map、栈、数据包缓冲区上下文）进行 1-8 字节的加载/存储；前/后（非）条件跳转；算术/逻辑操作；函数调用等

Profiling 工具

perf　　eBPF（bcc、bpftrace）　Intel Processor Trace

K/UProbe

1）替换目标指令为断点指令BREAKPOINT

2）断点异常

3）中断处理（检查kprobe注册表，调用pre_handler）

4）设置单步模式执行原有指令

5）完成后单步异常

6）调用post_handler

7）正常运行

（Linux text_poke机制）

Kretprobe （入口Kprobe+返回跳转Kprobe ）

数百ns ~ 千ns 时序操作

Tracepoint

1）内核编译时，通过DECLARE_TRACE，在tracepoint位置处留一个5Bytes的nop指令（x86），后续可静态替换为jump

2）函数尾加上tracepoint handler (trampoline)，运行时用于扫描tracepoint handler数组看是否有注册的

3）runtime当使能这个tracepoint时，增加注册函数，

4）nop会被替换为jump，跳到trampoline，运行注册函数

（Linux Jump Label/static-key、 text_poke机制 ）

数十ns ~ 数百ns 时序操作

Profiling 工具

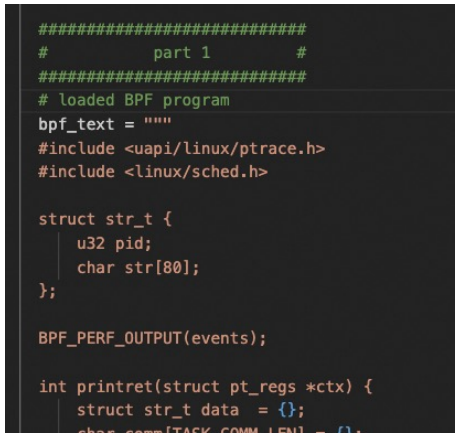
perf　　eBPF（bcc、bpftrace）　Intel Processor Trace

eBPF 程序的高层次组件

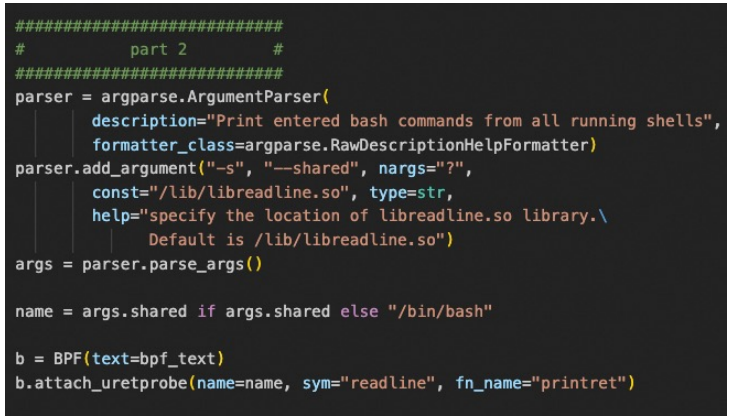
BCC：

*后端和数据结构：*用 “限制性 C” 编写。可以在单独的文件中，或直接作为多行字符串存储在加载器/前端的脚本中；

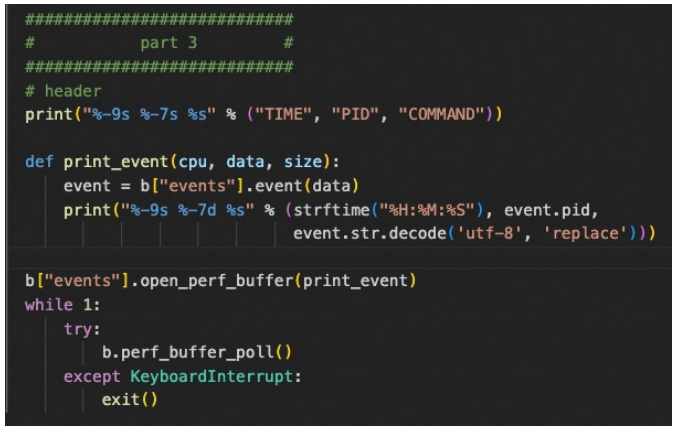
*加载器和前端：*可用简单的高级语言 python/lua 脚本编写。



```
#########################
#          part 1       #
#########################
# loaded BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct str_t {
    u32 pid;
    char str[80];
};

BPF_PERF_OUTPUT(events);

int printret(struct pt_regs *ctx) {
    struct str_t data  = {};
```



```
#########################
#          part 2       #
#########################
parser = argparse.ArgumentParser(
        description="Print entered bash commands from all running shells",
        formatter_class=argparse.RawDescriptionHelpFormatter)
parser.add_argument("-s", "--shared", nargs="?",
        const="/lib/libreadline.so", type=str,
        help="specify the location of libreadline.so library.\
              Default is /lib/libreadline.so")
args = parser.parse_args()

name = args.shared if args.shared else "/bin/bash"

b = BPF(text=bpf_text)
b.attach_uretprobe(name=name, sym="readline", fn_name="printret")
```



```
#########################
#          part 3       #
#########################
# header
print("%-9s %-7s %s" % ("TIME", "PID", "COMMAND"))

def print_event(cpu, data, size):
    event = b["events"].event(data)
    print("%-9s %-7d %s" % (strftime("%H:%M:%S"), event.pid,
                            event.str.decode('utf-8', 'replace')))

b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

Bpftrace：

建立在 BCC 之上，可以作为在寻找 BCC 的全部功能之前的快速分析/调试使用

bpftrace -e 'tracepoint:raw_syscalls:sys_enter {@[pid, comm] = count();}'

Profiling 工具

perf　　eBPF（bcc、bpftrace）　Intel Processor Trace

```
# 检测disk的I/O状态
biolatency -D [-Q include OS queued time in I/O time]
# fs层面
ext4dist / xfsdist -p 123
# 追踪pid程序fs操作大于10ms的操作
ext4slower / xfsslower 10 -p 123
# function count/latency/call-interval统计:
funccount/funclatency/funcinterval [-p PID] [-i INTERVAL] [-d DURATION] [-T] [-r] [-c CPU]
# 采集on-CPU火焰图
 sudo ./bcc/tools/profile -p $(pgrep mysqld) -f 3 --stack-storage-size=165535 > out.profile01.txt
 flamegraph.pl --width=1600 < out.profile01.txt > out.profile01.svg
# 采集off-CPU火焰图
 sudo ./bcc/tools/offcputime -p $(pgrep mysqld) -f 3 --stack-storage-size=165535 > out.offcpu01.txt
 flamegraph.pl --width=1600 --color=io --countname=us < out.offcpu01.txt > out.offcpu01.svg
# 采集pid程序page-fault超过的1线程及其1次的次数
bpftrace -e 'software:faults:1 /pid==12345/ { @[comm, tid] = count(); }'
# 采集page-fault的user-level stack
bpftrace -e 'tracepoint:exceptions:page_fault_user { @[ustack, comm] = count(); }'
```

Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

Intel Processor Trace（IPT）是英特尔处理器中引入硬件级指令追踪技术，它通过硬件高效记录控制流变化（如分支、中断、异常等），为调试、性能分析、安全监控等场景提供深度支持。

Change of Flow Instruction (COFI) Tracing

Table 33-1. COFI Type for Branch Instructions

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2), RET (C3, C2 xx)
Far Transfers	INT1, INT3, INT *n*, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

ns 级别时序操作

Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

Packet Stream Boundary (PSB) packets：周期性插入的同步包，用于数据流分割和错误恢复，提供解码起始点；

Time-Stamp Counter (TSC) packets： 记录 wall-clock time，各个core独立，特定事件触发比如 PSB；
Mini Time Counter (MTC) packets：周期的提供粗粒度的 wall-clock time，较为高频；
（TMA = TSC/MTC Alignment packets：可以对齐 TSC 和 MTC）

Cycle Count (CYC) packets： 记录 processor core 经过的 clock cycles，在周期精确模式下提供；
Core Bus Ratio (CBR) packets：记录 core/bus 的比例，可以把 core 周期转换为时钟芯片周期；

Taken Not-Taken (TNT) packets: 记录 direct conditional branches 的选择方向；
Target IP (TIP) packets：记录 indirect branches, interrupts 的目标 IP 方向；
Flow Update Packets (FUP):在异常、中断或特定事件后记录源 IP 地址；

PTWRITE (PTW) packets: 软件指令写入包内容

Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

Intel PT 的高层次使用程序

PERF-INTEL-PT:　　https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html

perf record -e intel_pt//　　perf {report | script} --itrace=

record 配置：intel_pt/配置1/配置2 配置other

配置1 **tsc** **mtc+mtc_period** **cyc** **noretcomp**（TIP when return） **branch**

psb_period /sys/bus/event_source/devices/intel_pt/caps/psb_periods

配置2 **u** userspace **k** kernel

--kcore 或 echo 'kernel.kptr_restrict=0' >> /etc/sysctl.conf

-m,16M { 提前配置perf事件锁定在RAM中的内存量 echo $[32*1024] > /proc/sys/kernel/perf_event_mlock_kb }

-S snapshot模式，无回环

--aux-sample -e branch-misses:u　　采样模式

--filter 'filter func @ /path/my_proc'

decode 配置：--itrace=配置（默认--itrace=cepwxy）

```
i       synthesize "instructions" events
y       synthesize "cycles" events
b       synthesize "branches" events
x       synthesize "transactions" events
w       synthesize "ptwrite" events
p       synthesize "power" events (incl. PSB events)
c       synthesize branches events (calls only)
r       synthesize branches events (returns only)
o       synthesize PEBS-via-PT events
I       synthesize Event Trace events
e       synthesize tracing error events
```

```
d       create a debug log
g       synthesize a call chain (use with i or x)
G       synthesize a call chain on existing event records
l       synthesize last branch entries (use with i or x)
L       synthesize last branch entries on existing event records
s       skip initial number of events
q       quicker (less detailed) decoding
A       approximate IPC
Z       prefer to ignore timestamps (so-called "timeless" decoding)
```

Profiling 工具

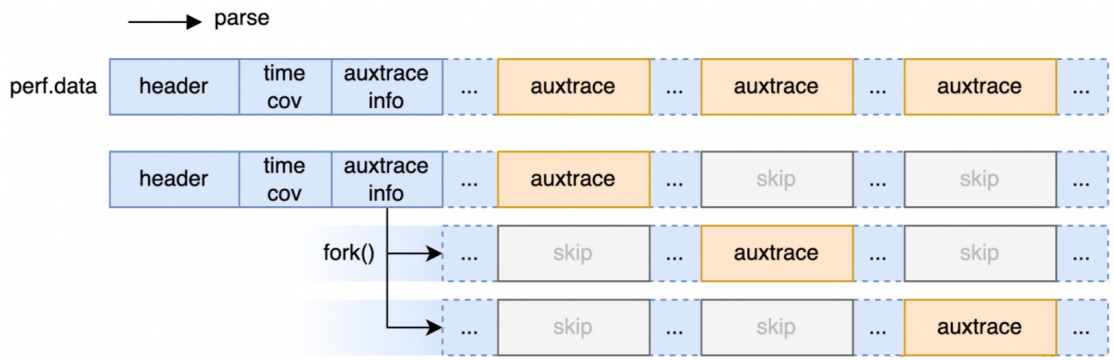
perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

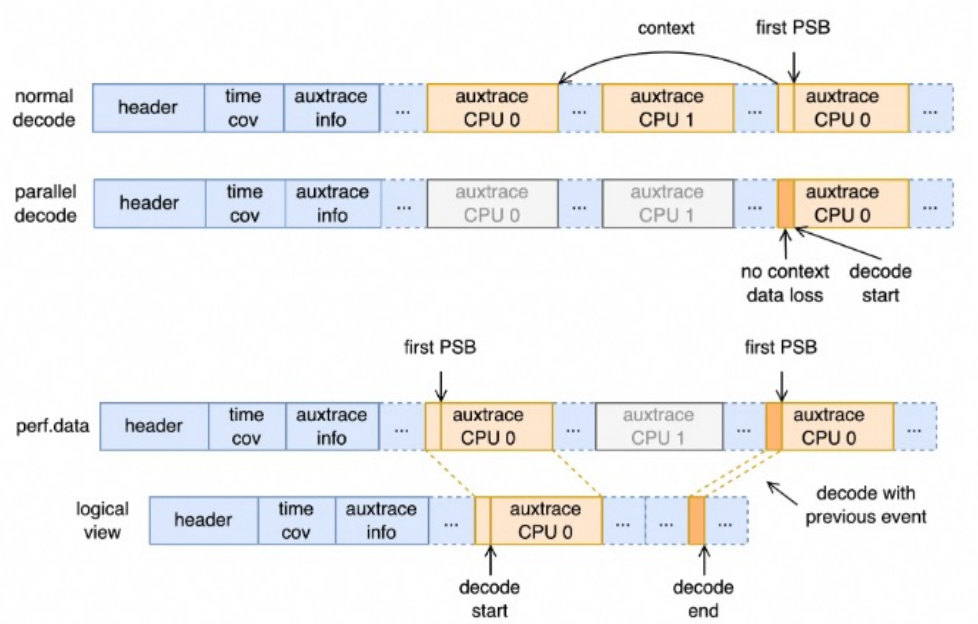
Intel PT 的高层次使用程序

并行perf script：

perf 以 event 形式从硬件目标 buffer 采集（copy）的 raw PT packages，并且封装成 AUXTRACE_INFO（含有所有auxtrace 数据的索引）、AUXTRACE（含有 PT packages）、AUX （匹配AUXTRACE含有状态信息）三种 perf reocrd 包。

依赖 PSB packets，decoder 可以开始 PT Stream packages 解析的（decoder 第一个包定位）。





* perf 在解析事件时可能会需要更前面的上下文信息，导致数据不够完整。
常见是推导调用栈信息时可能会出现调用栈丢失栈底的情况

Profiling 工具

perf　　eBPF（bcc、bpftrace）　　Intel Processor Trace

Intel PT 的高层次使用程序

ptfold_stack（函数/stack 分布统计）
pt_flame (火焰图生成)

```
call    log_flush_notifier          os_event_wait_time_low
jmp     os_event_wait_time_low      os_event::wait_time_low
jcc     os_event::wait_time_low     os_event::wait_time_low
call    os_event::wait_time_low     ut_usectime
...
```

pt_func_perf（整合，单函数）： ***install pt_func_perf***　　https://github.com/mysqlperformance/pt_perf/blob/main/README_CN.md

```
./func_latency -b "mysqld" -f "do_command" -d 1 -T tid1,tid2 -t -s [-i -o]
./func_latency -b "mysqld" -f "do_command" -d 1 -p pid -t -s [-i -o]
-i              需要510以上内核，开启ip_filter功能
-o              查看函数执行时 oncpu 和 offcpu 时延比例
-t              perf 使用 per_thread模式
-s              使用并发script

. /func_latency -b "mysqld" -f "do_command" -d 5 -T tid -t -s --history=1
./func_latency -b "mysqld" -f "do_command" -d 1 -T tid -t -s [-o] --history=2
--history=1 采集perf.data; --history=2使用perf.data

. /func_latency -b "mysqld" -f "trx_commit" -d 5 -T tid -t -s -l [--tu=100] --history=2
-l              函数时延的时间线timeline功能
--tu            每多少次取latency平均，默认是1


./func_latency --flamegraph="latency" -d 1 -p pid -t -s #latency火焰图
./func_latency --flamegraph="CPU" -d 1 -p pid -t -s  #cpu火焰图
```

```
================================================================================
Histogram - Latency of [parent_func]:
        ns          : cnt       distribution
       128 -> 255       : 1       |                    |
       256 -> 511       : 91443   |********************|
       512 -> 1023      : 66795   |**************      |
      1024 -> 2047      : 14746   |***                 |
      2048 -> 4095      : 10695   |**                  |
      4096 -> 8191      : 1310    |                    |
      8192 -> 16383     : 1147    |                    |
     16384 -> 32767     : 215     |                    |
     32768 -> 65535     : 36      |                    |
trace count: 186388, average latency: 869 ns
--------------------------------------------------------------------------------
Histogram - Child functions's Latency of [recv_multi_rec]:
            name            : avg       cnt       distribution (total)
func 1                      : 2383      2778      |*                   |
func 2                      : 262       181       |                    |
func 3                      : 184       397439    |********************|
func 4                      : 139       46866     |*                   |
func 5                      : 120       46865     |*                   |
func 6                      : 64        2022      |                    |
func 7                      : 30        1356404   |**********          |
================================================================================
```