Bpftrace

Terminologies

- Static Tracing
  - tracepoints are put explicitly into the source code, the tracing framework can then enable or disable those tracepoints at run time as desired
- Dynamic Tracing
  - tracepoints are injected into a running system, usually in the form of a breakpoint instruction.

Background

Introduction

bpftrace is a high-level tracing language for Linux enhanced Berkeley Packet Filter (eBPF) available in recent Linux kernels (4.x). bpftrace uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints.

- able to trace kprobes / uprobes / tracepoints
- better alternatives to ftrace / perf

Installation

- from pre-built binary in docker (recommended)

```
$ docker pull quay.io/iovisor/bpftrace:master-vanilla_llvm_clang_glibc2.23
$ docker run -v $(pwd):/output quay.io/iovisor/bpftrace:master-vanilla_llvm_clang_glibc2.23 /bin/bash -c
"cp /usr/bin/bpftrace /output"
$ sudo mv bpftrace /usr/local/bin
```

- from snap for Ubuntu 18.04

```
$ sudo snap install --devmode bpftrace
```

- from apt for Ubuntu 20.04

```
$ sudo apt install bpftrace
```

Usage

1. One-liner

```
$ sudo bpftrace -e 'kprobe:do_sys_open /comm == "vim"/ { printf("open %s\n", str(arg1)); }'
Attaching 1 probe...
open ~/.vimrc
open ~/.viminfo
open /etc/hosts
...
```

2. **Script**

```
$ cat trace_open.bt
kprobe:do_sys_open /comm == "vim"/ {
  printf("open %s\n", str(arg1));
}

$ sudo ./trace_open.bt
Attaching 1 probe...
open ~/.vimrc
open ~/.viminfo
open /etc/hosts
...
```

Syntax

- **probe[,probe,...] /filter/ { action }**
  - e.g: kprobe:do_sys_open /comm == "vim"/ { printf("open %s\n", str(arg1)); }
- Probes

kprobe	kernel function start	
kretprobe	kernel function return	
uprobe	user-level function start	
uretprobe	user-level function return	
tracepoint	kernel static tracepoint	TRACE_EVENT()
usdt	user-level static tracepoint	DTRACE_PROBE()
profile	timed sampling	
interval	timed output	
software	kernel software events	cpu-clock / page-faults / context-switches / ...
hardware	processor-level events	cpu-cycles / cache-references / cache-misses / ...

```
$ sudo bpftrace -l
...
kprobe:do_sys_open
...
tracepoint:syscalls:sys_enter_open
...
hardware:cache-misses:
...
software:page-faults:
...
```

- Builtin Variables

pid	process id
comm	process name
kstack	kernel call stack
arg0, arg1, ..., argN	arguments for kprobe
args	arguments for tracepoint
retval	return value for kretprobe

- Basic Variables

@global_name	visible to all probes
@associative_array_name[key_name]	visible to all probes
$scratch_name	visible to current probe

Example

kfd.bt (using kprobe)

```
#!/usr/bin/env bpftrace

#include "/usr/src/amdgpu-3.10-27/include/uapi/linux/kfd_ioctl.h"

k:kfd_ioctl_alloc_memory_of_gpu {
                @alloc_args = (struct kfd_ioctl_alloc_memory_of_gpu_args *)arg2;
                printf("%s:\n", func);
                printf("    va_addr: 0x%llx\n", @alloc_args->va_addr);
                printf("    size: 0x%llx\n", @alloc_args->size);
                printf("    mmap_offset: 0x%llx\n", @alloc_args->mmap_offset);
                printf("    flags: %s%s%s%s%s",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_VRAM ? " VRAM" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_GTT ? " GTT" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_USERPTR ? " USERPTR" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_DOORBELL ? " DOORBELL" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_MMIO_REMAP ? " MMIO_REMAP" : "");
                printf("%s%s%s%s%s\n",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_VRAM ? " VRAM" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_WRITABLE ? " WRITABLE" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_EXECUTABLE ? " EXECUTABLE" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_PUBLIC ? " PUBLIC" : "",
                                @alloc_args->flags & KFD_IOC_ALLOC_MEM_FLAGS_COHERENT ? " COHERENT" : "");
                printf("\n");
        }

kr:kfd_ioctl_alloc_memory_of_gpu {
                 printf("%s:\n", func);
                 printf("    handle: 0x%llx\n", @alloc_args->handle);
                 printf("    mmap_offset: 0x%llx\n", @alloc_args->mmap_offset);
                 printf("\n");
         }
```

```
$ ./hsatest --gtest_filter=HSAMemoryTest.UserMemoryAllocation

$ sudo ./amdkfd.bt
Attaching 2 probes...
kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617d06000
    size: 0x1000
    mmap_offset: 0x0
    flags:  MMIO_REMAP WRITABLE COHERENT

kretprobe_trampoline (21 us):
    handle: 0x44d300000002
    mmap_offset: 0x1134c00000000000

kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617cf8000
    size: 0x8000
    mmap_offset: 0x0
    flags:  GTT WRITABLE EXECUTABLE COHERENT

kretprobe_trampoline (26 us):
    handle: 0x44d300000003
    mmap_offset: 0x100c28000

kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617d03000
    size: 0x1000
    mmap_offset: 0x7f1617d03000
    flags:  USERPTR WRITABLE EXECUTABLE COHERENT

kretprobe_trampoline (23 us):
    handle: 0x44d300000004
    mmap_offset: 0x100c30000

kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617cf4000
    size: 0x2000
    mmap_offset: 0x7f1617cf4000
    flags:  USERPTR WRITABLE EXECUTABLE COHERENT

kretprobe_trampoline (14 us):
    handle: 0x44d300000005
    mmap_offset: 0x100c31000

kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617d01000
    size: 0x1000
    mmap_offset: 0x7f1617d01000
    flags:  USERPTR WRITABLE EXECUTABLE COHERENT

kretprobe_trampoline (10 us):
    handle: 0x44d300000006
    mmap_offset: 0x100c33000

kfd_ioctl_alloc_memory_of_gpu:
    va_addr: 0x7f1617cf2000
    size: 0x1000
    mmap_offset: 0x55e9d15d9000
    flags:  USERPTR WRITABLE EXECUTABLE

kretprobe_trampoline (9 us):
    handle: 0x44d300000007
    mmap_offset: 0x100c340
```

hsa.bt (using uprobe)

```
#!/usr/bin/env bpftrace

enum {
  HSA_QUEUE_TYPE_MULTI = 0,
  HSA_QUEUE_TYPE_SINGLE = 1,
  HSA_QUEUE_TYPE_COOPERATIVE = 2
}

u:/opt/rocm/lib/libhsa-runtime64.so.1:hsa_queue_create {
     printf("%s by %s(%d):\n", func, comm, pid);
     printf("    queue type: %s\n", arg2 == HSA_QUEUE_TYPE_MULTI ? "Multi" :
         (arg2 == HSA_QUEUE_TYPE_SINGLE ? "Single" :
          (arg2 == HSA_QUEUE_TYPE_COOPERATIVE ? "Cooperative" : "Unknown")));
     print(ustack);
   }
```

```
$ hsatest --gtest_filter=HSAQueueTest.CreateQueue

$ sudo ./hsa.bt
Attaching 1 probe...
hsa_queue_create by hsatest(19104):
    queue type: Single

        hsa_queue_create+0
        HSAQueueTest_CreateQueue_Test::TestBody()+73
        0x5603da9d28e6
        0x5603da9cd5e5
        0x5603da9b8d12
        0x5603da9b9490
        0x5603da9b9a2c
        0x5603da9bec17
        0x5603da9d3c19
        0x5603da9ce5e7
        0x5603da9bd9d3
        0x5603da9f6401
        0x7fc60b609bf7
        0x2bd6258d4c544155
```

References

- https://github.com/iovisor/bpftrace
- https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md
- http://www.brendangregg.com/ebpf.html
- https://leezhenghui.github.io/linux/2019/03/05/exploring-usdt-on-linux.html