bpfTrace – Finally DTrace Replacement on Linux

Is not that great?

Peter Zaitsev, CEO Percona
March 7, 2020

Scale 18x
Pasadena, CA
Question

Who is Familiar with Dtrace?
Question

Who is familiar with eBPF?
About the Presentation

Dtrace and eBPF Refresher

eBPF Tools Landscape

Look at eBPFTrace
Instrumentation Basics
Observability

Being Able to See Inside Running System

Critical for System Operation Monitoring, Troubleshooting, Performance Optimization

Achieved through Instrumentation
Instrumentation

There is Static instrumentation and Dynamic instrumentation.

Capturing Information from the Running system
The Instrumentation Approach

"Tracing"
- Emitting event when particular code point is reached

"Sampling"
- Checking the system state (i.e., program stack) at periodic interval
## Static vs Dynamic Instrumentation

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Counters, Logging points etc placed throughout the code</td>
<td>• Dynamically chose what you want to instrument with running system</td>
</tr>
<tr>
<td>• Need to be mindful about overhead</td>
<td>• Dynamically change level of instrumentation</td>
</tr>
<tr>
<td>• Limited Depth</td>
<td>• Can go very deep</td>
</tr>
</tbody>
</table>
DTrace
DTrace

Dynamic Tracing Framework

Developed Sun Microsystems starting 2001

Released in Solaris 10 in 2005

Define Specific Tracepoints in Kernel and User Land

Trace Function Calls and More

No Overhead than not enabled

D Language (Inspired by C and Awk)
## DTrace Beyond Solaris

<table>
<thead>
<tr>
<th>Platform</th>
<th>Year Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacOS 10.5</td>
<td>2007</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>2008</td>
</tr>
<tr>
<td>NetBSD</td>
<td>2010</td>
</tr>
<tr>
<td>Oracle Linux</td>
<td>Supported since 2011</td>
</tr>
<tr>
<td>Code Re-Licensed GPLv2+</td>
<td>by Oracle in 2017</td>
</tr>
<tr>
<td>Dtrace is Coming to Windows</td>
<td></td>
</tr>
</tbody>
</table>
DTrace on Linux

Is not available in stock Linux Kernel

Is not available from major Linux Distributions

Recent GPL Code release is likely too little too late
Tracing in Linux
Many competing tracing frameworks and frontends rather than single one
Linux Tracing in Pictures

Linux tracing systems & how they fit together

Data sources:
- Kprobes (kernel functions)
- Uprobes (userspace C functions)
- Kernel tracepoints
- USDT/dtrace probes
- LTTng userspace tracing

Source: https://jvns.ca/blog/2017/07/05/linux-tracing-systems/
## Linux Tracing Infrastructure

<table>
<thead>
<tr>
<th>The Type of Kernel Interface</th>
<th>The Type of “Program” Connected to it</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Kprobe, uprobe, Dtrace probe etc</td>
<td>• Built in Kernel Buffer, Kernel Module, eBPF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Front-end Tools to work with it from the user space</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Perf, SystemTap, SysDig, Bcc etc</td>
<td></td>
</tr>
</tbody>
</table>
eBPF – emerging Linux Standard
eBPF - Extended Berkeley Packet Filter

- Berkeley Packet Filter - Originated in 1992 as efficient virtual machine for Packet Filtering
- Extended Berkeley Packet Filter – Extended Version found in Linux
- General Event Processing Framework
- JIT Compiler for high efficiency
eBPF vs BPF

**BPF**
- 32-bit registers
- 16 x 32-bit memory
  - Accumulator
  - Scratch pad

**eBPF**
- 64-bit registers
  - r0
  - r1
  - r10
- Stack
  - 512 bytes
- Maps

© 2020 Percona.
eBPF in Linux

- Has been in Linux Kernel since 2014
- Actively being improved
- Integrated in “perf” tooling system
# Improvements in recent Kernels

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel Version</th>
<th>Commit SHA</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bpf2bpf</code> function calls</td>
<td>4.16</td>
<td>ccBb0b92a169</td>
</tr>
<tr>
<td>BPF used for monitoring RX/TX data</td>
<td>4.17</td>
<td>4F738adca30a</td>
</tr>
<tr>
<td>BPF attached to <code>bind()</code> system call</td>
<td>4.17</td>
<td>4Fbac77d2d09</td>
</tr>
<tr>
<td>BPF Type Format (BTF)</td>
<td>4.18</td>
<td>69b693f0aefa</td>
</tr>
<tr>
<td>AF_XDP</td>
<td>4.18</td>
<td>Fbfc504a24f5</td>
</tr>
<tr>
<td><code>bpfilter</code></td>
<td>4.18</td>
<td>d2ba09c17a06</td>
</tr>
<tr>
<td>End.BPF action for seg6local LWT</td>
<td>4.18</td>
<td>0eb4d4b274e2a</td>
</tr>
<tr>
<td>BPF attached to LIRC devices</td>
<td>4.18</td>
<td>F4364d8c86d</td>
</tr>
<tr>
<td>BPF socket reuseport</td>
<td>4.19</td>
<td>2dbbb9e6df6</td>
</tr>
<tr>
<td>BPF flow dissector</td>
<td>4.20</td>
<td>d58e468b1112</td>
</tr>
<tr>
<td>BPF cgroup sysctl</td>
<td>5.2</td>
<td>7b146cebe30c</td>
</tr>
<tr>
<td>BPF raw tracepoint writable</td>
<td>5.2</td>
<td>9df1c28bb752</td>
</tr>
</tbody>
</table>

eBPF Programs

- Linux Kernel can load programs in custom byte code
- Programs verified before load to prevent misuse
- LLVM Clang can compile to eBPF byte code
- This compilation is kernel-dependent
- Few will need to write eBPF programs Directly
eBPF User Space vs Kernel

Source: http://www.brendangregg.com/ebpf.html
eBPF features in different kernel versions

http://www.brendangregg.com/ebpf.html#2017
Project to Know

https://github.com/iovisor
eBPF Overhead

eBPF Programs can be run million+ times per second per core

<table>
<thead>
<tr>
<th>Case</th>
<th>ns/op</th>
<th>overhead ns/op</th>
<th>ops/s</th>
<th>overhead percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>no probe</td>
<td>316</td>
<td>0</td>
<td>3,164,556</td>
<td>0%</td>
</tr>
<tr>
<td>simple</td>
<td>424</td>
<td>108</td>
<td>2,358,490</td>
<td>34%</td>
</tr>
<tr>
<td>complex</td>
<td>647</td>
<td>331</td>
<td>1,545,595</td>
<td>105%</td>
</tr>
</tbody>
</table>

eBPF Frontends
The eBPF Tracing Landscape, Jan 2019

(Brendan’s opinion)

Ease of use (less brutal)

(alpha) → (mature)

Stage of Development

Scope & Capability

perf

ftrace

perf BPF

bcc

stap BPF

ply

bpctrace

C BPF

Raw BPF
Most Valuable

BCC
- Has Great set of Pre-Built tools
- Tricky to Develop your own Tools

BpfTrace
- Much Easier to use Language
- Collection of Tools is being Built out
BCC

from __future__ import print_function
from bcc import BPF
from time import strftime
import ctypes as ct

# load BPF program
bpf_text = ---
#include <uapi/linux/ptrace.h>
struct str_t {
    u64 pid;
    char str[80];
};
BPF_PERF_OUTPUT(events);
int printret(struct pt_regs *ctx) {
    u32 pid;
    if (PT_REGS_RC(ctx))
        return 0;
    pid = bpf_get_current_pid_tgid();
    data.pid = pid;
    bpf_probe_read(&data.str, sizeof(data.str), (void *)PT_REGS_RC(ctx));
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}

class Data(ct.Structure):
    _fields_ = [
        ('pid', ct.c_ulonglong),
        ('str', ct.c_char * STR_DATA)
    ]

b = BPF(text=bpf_text)
b.attach_uretprobe(name='/bin/bash', sym='readline', fn_name='printret')

# header
print('%-9s %-6s %s' % ('TIME', 'PID', 'COMMAND'))
def print_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data)).contents
    print('%-9s %-6d %s' % (strftime('%H:%M:%S'), event.pid,
        event.str.decode('utf-8', 'replace')))
b['events'].open_perf_buffer(print_event)
while 1:
    b.perf_buffer_poll()

bpfftrace

BEGIN
{
    printf("Tracing bash commands... Hit Ctrl-C to end.\n");
    printf("%-9s %-6s %s\n", "TIME", "PID", "COMMAND");
}

uretprobe:/bin/bash:readline
{
    time("%H:%M:%S ");
    printf("%-6d %s\n", pid, str(retval));
}
BCC Tools Available

https://github.com/iovisor/bcc#tools 2018
DTrace vs bpfTrace
General Landscape Comparison

bpfTrace and DTrace

There is no Direct compatibility

Similar in Spirit

bpfTrace is more powerful
# Function Comparison Checklist

<table>
<thead>
<tr>
<th>Type</th>
<th>DTrace</th>
<th>bptrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>$@ = quantize(value)</td>
<td>$@ = hist(value)</td>
</tr>
<tr>
<td>function</td>
<td>$@ = lquantize(value, min, max, step)</td>
<td>$@ = lhist(value, min, max, step)</td>
</tr>
<tr>
<td>variable</td>
<td>this-&gt;name</td>
<td>$name</td>
</tr>
<tr>
<td>variable</td>
<td>self-&gt;name</td>
<td>@name[tid]</td>
</tr>
<tr>
<td>variable</td>
<td>name[key]</td>
<td>@name[key]</td>
</tr>
<tr>
<td>variable</td>
<td>global_name</td>
<td>@global_name</td>
</tr>
<tr>
<td>variable</td>
<td>self-&gt;name = 0</td>
<td>delete(@name[tid])</td>
</tr>
<tr>
<td>variable</td>
<td>curthread</td>
<td>curtask</td>
</tr>
<tr>
<td>variable</td>
<td>probeprov probemod probename probe</td>
<td></td>
</tr>
<tr>
<td>provider</td>
<td>fbt::func:entry</td>
<td>kprobe:func</td>
</tr>
<tr>
<td>provider</td>
<td>fbt::func:return</td>
<td>kretprobe:func</td>
</tr>
<tr>
<td>provider</td>
<td>pidStarget::func:entry</td>
<td>uprobe:func</td>
</tr>
<tr>
<td>provider</td>
<td>pidStarget::func:return</td>
<td>uretprobe:func</td>
</tr>
<tr>
<td>provider</td>
<td>profile:::99</td>
<td>profile:hz:99</td>
</tr>
<tr>
<td>provider</td>
<td>profile:::tick-1sec</td>
<td>intervals:1</td>
</tr>
</tbody>
</table>
Script Example
bpfTrace
Linux Requirements

Kprobes (4.1) kprobe:vfs_read { ... }
Uprobes (4.3) uprobe:/bin/bash:readline { ... }
USDT (4.3)
Stack traces, per-cpu maps (4.6)
Tracepoints (4.7) tracepoint:sched:sched_switch { ... }
Timers (4.9) profile:hz:99 { ... }
Software events (4.9) software:faults: { ... }
Hardware events (4.9) hardware:cache-references: { ... }

bpfTrace Probe Types

Dynamic Tracing

**tracepoint:** Static Tracing

- ext4
- Operating System
- Applications
- System Libraries
- System Call Interface
- Sockets
- TCP/UDP
- Volume Manager
- IP
- Ethernet
- Virtual Memory
- Scheduler
- Device Drivers
- jbd2
- Block Device Interface
- skb
- net

**uprobe:**
- uprobe
- usdt

**kprobe:**
- kprobe
- kretprobe

**BEGIN**
- Special Events

**END**

**software:**
- cpu-clock
- cs
- migrations
- page-faults
- minor-faults
- major-faults

**hardware:**
- cpu-cycles
- instructions
- branch-*
- frontend-*
- backend-*
- bus
- Memory Bus
- cache-*

**profile:**
- interval:

Timed Events

https://github.com/iovisor/bpftrace

© 2020 Percona.
How BPFTrace Works

bptrace Internals

Kernel

Events:
- tracepoints
- kprobes
- uprobes
- perf_events

BPF

Verifier

Maps

Async Summaries

Per-event Output, Async Actions
- printf()
- AsyncAction::*

BPF bytecode

IR Builder

Code Generation
- ast/codegen/lstmt.*
- AST Nodes to LLVM IR calls
- BPF func calls

Semantic Analyzer

Clang Parser

Parser

- clangParser.*
- parse bpfttrace program into AST

AST

process structs

create_maps()

bptrace_add_probe()

Maps

- maps.*
- maps_

Probes

- probes
- probe

Attached Probes

attached_probes

print_map()

perf_event_printer()

BPF

bpffl_load_program()

bpffl_attach_*(), bpffl_usdt_enable_probe()

Async Actions

https://github.com/visor/bpftrace 2018
Support In Linux Distributions

Not all Distributions have packages

Development is Quick Paced – Many have outdated packages

If you use eBPF consider getting new packages
Install bpftrace

Ubuntu snap package

Warning: snap packages have limited functionality

On Ubuntu 16.04 and later, bpftrace is available as a snap package (https://snapcraft.io/bpftrace) and can be installed with snap. The current snap provides extremely limited file permissions so the --devmode option should be specified on installation in order avoid file access issues.

```bash
sudo snap install --devmode bpftrace
sudo snap connect bpftrace:system-trace
```

More Details: https://github.com/iovisor/bpftrace/blob/master/INSTALL.md
Timing Reads by Process

```bash
bpftrace -e 'kprobe:vfs_read { @start[tid] = nsecs; }
kretprobe:vfs_read /@start[tid]/ { @ns[comm] = hist(nsecs - @start[tid]); delete(@start[tid]); }'
```

<table>
<thead>
<tr>
<th>Size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>512, 1K</td>
<td>8</td>
</tr>
<tr>
<td>1K, 2K</td>
<td>1755</td>
</tr>
<tr>
<td>2K, 4K</td>
<td>3237</td>
</tr>
<tr>
<td>4K, 8K</td>
<td>504</td>
</tr>
<tr>
<td>8K, 16K</td>
<td>142</td>
</tr>
<tr>
<td>16K, 32K</td>
<td>42</td>
</tr>
<tr>
<td>32K, 64K</td>
<td>12</td>
</tr>
<tr>
<td>64K, 128K</td>
<td>17</td>
</tr>
<tr>
<td>128K, 256K</td>
<td>22</td>
</tr>
<tr>
<td>256K, 512K</td>
<td>505</td>
</tr>
<tr>
<td>512K, 1M</td>
<td>315</td>
</tr>
<tr>
<td>1M, 2M</td>
<td>61</td>
</tr>
<tr>
<td>2M, 4M</td>
<td>41</td>
</tr>
<tr>
<td>4M, 8M</td>
<td>37</td>
</tr>
<tr>
<td>8M, 16M</td>
<td>11</td>
</tr>
<tr>
<td>16M, 32M</td>
<td>2</td>
</tr>
<tr>
<td>32M, 64M</td>
<td>1</td>
</tr>
</tbody>
</table>
Saving it as Script File

```c
// read.bt file
tracepoint:sysevents:sys_exit_read
{
    @start[tid] = nsecs;
}

tracepoint:sysevents:sys_exit_read / @start[tid] /
{
    @times = hist(nsecs - @start[tid]);
    delete(@start[tid]);
}
```

# bpftrace read.bt
Attaching 2 probes...
^C
## Concept Overview

### General Syntax
- `probe[,,probe,...] /filter/ { action }`

### Filter
- Filtering output of the Probe (ie by Pid)

### Action
- Mini-Program to be ran

https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md
BpfTrace Tools

Tools

bpftrace contains various tools, which also serve as examples of programming in the bpftrace language.

- tools/bashreadline.bt: Print entered bash commands system wide. Examples.
- tools/biolatency.bt: Block I/O latency as a histogram. Examples.
- tools/biosnoop.bt: Block I/O tracing tool, showing per I/O latency. Examples.
- tools/biostacks.bt: Show disk I/O latency with initialization stacks. Examples.
- tools/bitesize.bt: Show disk I/O size as a histogram. Examples.
- tools/capable.bt: Trace security capability checks. Examples.
- tools/cpuwalk.bt: Sample which CPUs are executing processes. Examples.
- tools/dcsnoop.bt: Trace directory entry cache (dcache) lookups. Examples.
- tools/killsnopo.bt: Trace signals issued by the kill() syscall. Examples.

https://github.com/iovisor/bpftrace
Curios to see BPF Code?

```c
# bpftrace -v -e 'tracepoint:syscalls:sys_enter_nanosleep { printf("%s is sleeping.\n", comm); }'
Attaching 1 probe...

Bytecode:
0: (bf) r6 = r1
1: (b7) r1 = 0
2: (7b) *(u64 *)(r10 -24) = r1
3: (7b) *(u64 *)(r10 -32) = r1
4: (7b) *(u64 *)(r10 -40) = r1
5: (7b) *(u64 *)(r10 -8) = r1
6: (7b) *(u64 *)(r10 -16) = r1
7: (bf) r1 = r10
8: (07) r1 += -16
9: (b7) r2 = 16
10: (85) call bpf_get_current_comm#16
11: (79) r1 = *(u64 *)(r10 -16)
12: (7b) *(u64 *)(r10 -32) = r1
```
Tracing MySQL

bpftrace -e 'uprobe:/usr/sbin/mysqld:dispatch_command
{ printf("%s\n", str(arg2)); }'
failed to stat uprobe target file /usr/sbin/mysqld: No such file or directory

root@mysql1://# ls -la /usr/sbin/mysqld
-rwxr-xr-x 1 root root 60718384 Oct 25 09:19 /usr/sbin/mysqld
Tracing MySQL

Using apt installed bpftrace rather than snap package

root@mysql1:~# bpftrace -e 'uprobe:/usr/sbin/mysqld:dispatch_command { printf("%s\n", str(arg2)); }'
Attaching 1 probe...
Could not resolve symbol: /usr/sbin/mysqld:dispatch_command
Tracing MySQL(MariaDB)

root@mysql1:~# nm -D /usr/sbin/mysqld | grep dispatch_command
00000000005af770 T
_Z16dispatch_command19enum_server_commandP3THDPcjbb
root@localhost:~# bpftrace -e 'uprobe:/usr/sbin/mysqld:_Z16dispatch_command19enum_server_commandP3THDPcjbb
{ printf("%s\n", str(arg2)); }'
Attaching 1 probe...
select @@version_comment limit 1
select 1
Check out eBPF Bible

http://www.brendangregg.com/ebpf.html
Further Reading List

https://github.com/zoidbergwill/awesome-ebpf
https://slideplayer.com/slide/12710510/
http://www.brendangregg.com/ebpf.html
https://lwn.net/Articles/740157/
Percona Live is the one and only event where all of the open source database solution companies come together with the community. **MySQL, MongoDB, PostgreSQL, Elastic, Redis and more...**

*Percona Live brings them to you!*

- 3 Days
- Hands-on tutorials
- Breakout sessions
- Keynote addresses
- Expo Hall
- Networking
- Lots of Fun!

Use **CONFERENCE** for 20% off! Register now at perconalive.com
Thank You!