HOW I FELL IN LOVE WITH SYSTEMTAP AND PERF

RALPH BÖHME, SERNET, SAMBA TEAM

UNDERSTANDING AND IMPROVING SAMBA FILESERVER PERFORMANCE
AGENDA

- Disclaimer: focus on userspace, not kernel, mostly Linux
- Linux tracing history tour de force
- perf
- Systemtap
- Samba fileserver performance improvements
KEY TAKEAWAY...
...LINUX TRACING HAS EVOLVED...
INTRODUCTION

TRACING IN THE 90’S

ptrace

/proc
INTRODUCTION

TRACING TODAY

ftrace  perf_events  eBPF  SystemTap
LTTng   ktap       dtrace4linux  OEL DTrace  sysdig
How do we use these superpowers?

taken from Brendan Greggs presentation Performance Analysis with bcc/BPF
A LINUX TRACING TIMELINE

1990’s: Static tracers
2000: LTT + DProbes
2004: kprobes
2005: DTrace
2005: SystemTap
2005: LTTng
2008: ftrace
2009: perf_events
2009: Kernel tracepoints
2012: uprobes
2013: ktap
2014: sysdig
2014: eBPF
What can be done:

- Counting CPU events: cycles, branch misses, frontline stalls, ...
- Trace syscalls, but more efficiently
- Trace at the source code level by symbol (function name) or line number (both kernel and userspace)
- Provide stable tracepoint ABI (again kernel and userspace)
- Counting, statistics, latency, histograms...
- Some stuff (BPF, ftrace with hist-triggers, uprobes) requires newer kernels so might not be present on older systems
The whole zoo uses a smaller set of common in-kernel tracing frameworks:

1. **Static tracepoints**

2. **Dynamic tracepoints**: `kprobes` and `uprobes`

3. **perf_events**

4. **BPF** (previously also **Enhanced BPF**, aka eBPF)

All frameworks incur low overhead when enabled per tracepoint and zero overhead when not enabled – except `uprobes` and `USDT` which take a context switch when firing.
The types of events are:

- **CPU Hardware Events**: CPU performance monitoring counters (PMU, Performance Monitoring Unit), e.g., CPU cycles

- **CPU Software Events**: these are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, branch misses etc.

- **Tracepoint Events**: This are static kernel-level (SDT) or user-level (USDT) instrumentation points that are hardcoded in interesting and logical places in the kernel or applications

- **Dynamic Tracing**: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the *kprobes* framework. For user-level software, *uprobes*.

- **Timed Events**: commonly used for profiling
INTRODUCTION

ftrace  perf_events  eBPF  SystemTap

LTTng  ktap  dtrace4linux  OEL DTrace  sysdig
BPF/bcc, the new kid on the block:

- **BPF**: (enhanced) Berkeley Packet Filter with, the kernel framework
- **bcc**: BPF compiler collection

BPF originated as a technology for optimizing packet filters. If you run tcpdump with an expression (matching on a host or port), it gets compiled into optimal BPF bytecode which is executed by an in-kernel sandboxed virtual machine.

- Enhanced BPF (aka eBPF, but also just BPF) extended what this BPF virtual machine could do: allowing it to run on events other than packets, and do actions other than filtering.
INTRODUCTION

**ftrace**

- It's been mentioned as kernel hacker's best friend, built into the kernel and can consume all the mentioned kernel tracing frameworks
- Event tracing, with optional filters and arguments
- Until very recently not programmable and no built-in statistics support, changed with the addition of hist-triggers and BPF support
How to choose? For userspace, like Samba:

- Recommendation: choose perf for CPU profiling
- Systemtap for all the rest
- Look at the others when something is missing (unlikely) or you feel like it
- Keep an eye on BPF
perf_events
perf_events: a kernel subsystem(s) and a user-space tool

- Counting events & profiling with post-processing
- Not programmable and no builtin statistics and aggregations, though this changed recently
- It can instrument CPU performance counters (PMU), tracepoints, kprobes and uprobes
Linux profilers:

1. GNU gprof: requires special compilation
2. Valgrind Callgrind: sloooooooooooooooooow
3. oprofile, just didn't work in my environment so I looked at:
4. perf
Where do you get it?

# yum install perf
# apt-get install linux-tools

When profiling you will want symbols so also install *-debuginfo/ *.dbg package of profiled application

perf can do much more than profiling, but for me the key selling point is the text-based interactive interface to display the profile info:

# perf report
DEMO
SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live running kernel plus user-space applications.

The essential idea behind … systemtap … is to name events, and to give them handlers. Whenever a specified event occurs, the Linux kernel runs the handler.

You write the event handlers in the Systemtap script language which is C like with type inference, but safe with builtin runtime safety checks.
The script associates handlers with probes:

```plaintext
probe EVENT { HANDLER }
```

Several varieties of supported events:
begin, end, timer, syscalls, tracepoints, DWARF, perf_events

Handler can have filtering, conditionals, variables: primitive (numbers, strings), associative arrays, in kernel statistical aggregations

Many helper functions: printf, gettimeofday, ...

The script is translated to C

...continued on next slide...
...continued from previous slide...

- The C code is compiled to a kernel module
- The kernel loads the module and enables the probes, inserting jumps (kernel) or breakpoints (userspace)
- With DWARF debug symbols you can place probes on file.c:linenumber (kernel or user-space)
- Associative arrays, Statistics (aggregates)
- Probe handlers have access to execution context (variables, parameters)
SYSTEMTAP: LIST AVAILABLE STATIC PROBES

$ # DWARF debug symbols
$ stap -l 'kernel.function("*")' | wc -l
   54049

$ # kprobes, doesn't require debug symbols
$ stap -l 'kprobe.function("*")' | wc -l
   43792

$ # SDT, no debug symbols needed
$ stap -l 'kernel.trace("*")' | wc -l
    2203

$ # CPU PMU Hardware/Software
$ stap -l 'perf.*.*' | wc -l
     19

# man stapprobes
**SYSTEMTAP EXAMPLE: HELLO WORLD**

```bash
# cat hello_world.stp
probe begin {
    printf("Hello world!\n")
}

probe end {
    printf("\nGoodbye!\n")
}

# stap hello_world.stp
Hello world!
^C
Goodbye!
```
SYSTEMTAP EXAMPLE: SYSCALL WITH STATISTICS

```
# cat pwrite.stp
global bytes_written

probe begin {
    printf("Tracing, press ctrl-c to stop... ")
}

probe syscall.pwrite.return{
    if(pid() == target())
        bytes_written += $return
}

probe end {
    printf("\nTotal bytes written: %d\n", bytes_written)
}

# stap -x 18113 pwrite.stp
Tracing, press ctrl-c to stop... ^C
Total bytes written: 2825879
```
# cat smb2-reqs.stp

```plaintext
global smb2_reqs

probe begin {
    printf("Tracing, press ctrl-c to stop... ")
}

probe process("/.../libsmbd-base-samba4.so").function("smbd_smb2_request_dispatch") {
    if(pid() == target())
        smb2_reqs++
}

probe end {
    printf("\nGot %d SMB2 requests\n", smb2_reqs)
}
```

# stap smb2-reqs.stp

Tracing, press ctrl-c to stop... ^C

Got 7163 SMB2 requests
SYSTEMTAP EXAMPLE: ADDING USING USDT PROBE TO SAMBA

commit 3caa363dcf41aed3c2e4486d9f77880c3bb140f1
Author:     Ralph Boehme <slow@samba.org>

... s3/smbd: add instrumentation for SMB2 request
...
--- a/source3/smbd/smb2_create.c
+++ b/source3/smbd/smb2_create.c
... @@ -703,6 +705,8 @@ static void reprocess_blocked_smb2_lock(...)
     if (!smb2req->subreq) {
         return;
     }
+    SAMBA_PROBE(smb2, request_start, 2, smb2req->smb1req->mid, SMB2_OP_LOCK);
     SMBPROFILE_IOBYTES_ASYNC_SET_BUSY(smb2req->profile);
     state = tevent_req_data(smb2req->subreq, struct smbd_smb2_lock_state);
commit cad76c44f6f88caa08ff92d2dea73a120d4e9b59
Author: Ralph Boehme <slow@samba.org>
...
libreplace: add Systemtap include wrapper
...
--- /dev/null
+++ b/lib/replace/system/systemtap.h
@@ -0,0 +1,63 @@
+#ifdef HAVE_SYS_SDT_H
+#include <sys/sdt.h>
...
SYSTEMTAP EXAMPLE: USING USDT PROBE

```bash
# cat smb2-reqs2.stp
global smb2_reqs

probe begin {
    printf("Tracing, press ctrl-c to stop... ")
}

probe process("/.../libsmbd-base-samba4.so").provider("smb2").mark("request_start") {
    if(pid() == target())
        smb2_reqs++
}

probe end {
    printf("\nGot %d SMB2 requests\n", smb2_reqs)
}

# stap smb2-reqs2.stp
Tracing, press ctrl-c to stop... ^C
Got 8130 SMB2 requests
```
While at it, let’s also add instrumentation to these:

1. tevent events
2. sending / receiving data from the network
3. disk IO
4. syscalls
5. smbd ↔ ctdb communication latency

Let me introduce you to **tsmbd**: 
SYSTE MTAP: TSMBD

Trace an smbd process with Systemtap

Usage:

tsmbd [-s|-d|-h] pid

- pid       # trace this process ID
- d         # show distribution histograms
- h         # print this help text
SYSTEMTAP: TSMBD

# examples/systemtap/tsmbd 11327
Compiling Systemtap script, this may take a while...
Collecting data, press ctrl-C to stop... ^C
# examples/systemtap/tsmbd 11327
Compiling Systemtap script, this may take a while...
Collecting data, press ctrl-C to stop... ^C

Ran for:                          38728 ms
Time waiting for events:          32029 ms
Time receiving SMB2 packets:       157 ms
Time running SMB2 requests:        3820 ms
Time sending SMB2 packets:         832 ms
Time waiting for ctdb:              0 ms
Time in syscalls:                  2165 ms
Time in disk IO (read):            9 ms
Time in disk IO (write):           45 ms
Number of tevent events:          29407
Number of SMB2 requests:          26937
Number of ctdb requests:          0
### SMB2 Requests

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Count</th>
<th>Total ms</th>
<th>Avg us</th>
<th>Min us</th>
<th>Max us</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMB2_OP_CREATE</td>
<td>8295</td>
<td>2516071</td>
<td>303</td>
<td>65</td>
<td>13378</td>
</tr>
<tr>
<td>SMB2_OP_CLOSE</td>
<td>8152</td>
<td>573601</td>
<td>70</td>
<td>19</td>
<td>8218</td>
</tr>
<tr>
<td>SMB2_OP_SETINFO</td>
<td>5297</td>
<td>258329</td>
<td>48</td>
<td>19</td>
<td>8154</td>
</tr>
<tr>
<td>SMB2_OP_WRITE</td>
<td>2464</td>
<td>333957</td>
<td>135</td>
<td>62</td>
<td>8246</td>
</tr>
<tr>
<td>SMB2_OP_GETINFO</td>
<td>2729</td>
<td>125258</td>
<td>45</td>
<td>34</td>
<td>8222</td>
</tr>
</tbody>
</table>

### ctDB Requests

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Count</th>
<th>Total ms</th>
<th>Avg us</th>
<th>Min us</th>
<th>Max us</th>
</tr>
</thead>
</table>

...continued on next page...
 hypersocket (SMB2) request service time histogram (optional with tsmbd -d)...

SMB2_OP_CREATE distribution (microseconds)

<table>
<thead>
<tr>
<th>value</th>
<th>--------------------------------------------------</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>289</td>
</tr>
<tr>
<td>128</td>
<td>@@</td>
<td>846</td>
</tr>
<tr>
<td>256</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>9315</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>16384</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
tsmbd summary:

- Nice and detailed high level overview, it's non-intrusive
- tsmbd is only 356 lines of code, much of that is just boilerplate stuff for the probes
- Tracing too much details (like all syscalls) does have performance impact due to context switching, we may have to make that an option
- Another useful thing is to extend it to trace all SMB2 sessions, currently only one selected by process pid
1. Clustered Samba: directory enumeration

2. Name mangling: new option „mangled names = illegal“

3. Make use of struct smb_filename plumbing in the 4.5 VFS: avoid redundant stats

4. GPFS VFS module improvements: avoid GPFS API calls to fetch creation date

5. Internal messaging improvements: connection caching

6. Exclusive lease optimisations (Samba had this for oplocks but they didn't make it into the lease area): check file handle before looking into the leases database
Small file copy throughput:

- before: 136 files / s
- after: 151 files / s
- ~10% improvement by drilling into existing code with perf TUI
WIP Samba instrumentation git branch

Systemtap Beginners Guide

Systemtap Language Reference

Linux perf site by Brendan Gregg

BPF Compiler Collection

Flamegraphs
THANK YOU!
QUESTIONS?

Ralph Böhme <slow@samba.org>