Efficient and Large Scale Program Flow Tracing in Linux

Alexander Shishkin, Intel
16.09.2013
Overview

• Program flow tracing
  - What is it?
  - What is it good for?

• Intel® Processor Trace
  - Features / capabilities

• Linux support
  - Perf infrastructure
  - Use cases / scenarios
Program flow tracing
What is program flow trace?

• Branch history
  - As opposed to stack trace
  - As opposed to function trace only (ftrace)

• Function call graph

• Even better with timing information
What is it good for?

- Profiling / performance measurement
- Functional debugging
- Code coverage analysis
- Any other ideas?
Existing methods

- **In software (instrumentation, emulation, you name it)**
  - Intrusive, slow, doesn’t scale well

- **With hardware support**
  - x86: LBR/BTS
    - intrusive, limited timing information, non-architectural
  - ARM: ETM/PTM
  - PowerPC: BHRB
  - x86: Intel® Processor Trace
Features and capabilities

• Exact control flow information
• Mode related information
  - timing, paging, TSX state, execution mode, core-to-bus clock ratio
• Highly compressed packet output
• Filtering
  - Privilege level (CPL) / address space (CR3)
• Output to memory
Considerations

• Doesn’t require any modification to the code
  - Works with debug or production builds
• Can be used for system-wide tracing or JITted code
• Does require sideband information from the OS
  - Context switches, address space modifications, etc
• Also requires object code to be able to decode traces
• Non-zero performance overhead
Linux support
Use cases

- Profiling at very fine granularity
- Full trace mode
- Anomaly detection (snapshot) mode
- Sample annotation
- Process core dumps
- System core dumps
- “Flight recorder”
- GNU debugger support
Profiling at a very fine granularity

• Even down to a single instruction level
Full trace mode

- Userspace keeps collecting trace data
  - Kernel wakes it up
  - Trace stops if buffer fills up
  - Some data may get lost
Anomaly detection/snapshot mode

- Trace keeps running, but no data is collected from trace buffers
  - Overwriting older data
  - If an anomaly is detected, tracing is stopped briefly so that trace data can be collected
  - Otherwise tracing is stopped and trace data is discarded
- “something is taking too long”
- You tell us, what happened, we tell you how it happened
Sample annotation

- PT data can be used to annotate other perf events
  - Trace is retrieved every time a certain event takes place
  - Like a tracepoint or a PMU event
- Can be used to replace or complement for backtrace, providing more context
- To be used mostly with perf report
Core dumps

• Tracing for a process can be enabled via ulimit
  - If the process crashes, the most recent trace data for each thread is included in the core dump

• Tracing for the whole system can be enabled at boot time
  - If the system crashes, trace data with perf sideband data can be stored
    - in a EFI capsule
    - in a system crash dump
GNU debugger support

- Work is ongoing to enable PT (via perf interface) support in gdb
- Analyzing process core dumps
- Provide reverse execution
- Show control flow on assembly and function level
- Show or reverse-step through the code that led to a crash or transaction abort
GDB screenshot

(gdb) c
Continuing.

Breakpoint 2, find_charset_names () at charset.c:854
854     if (len <= 3)
(gdb) record function-call-history /cli -
1208    xmalloc inst 22429,22432 at ./common/common-utils.c:52,55
1209    xstrdup inst 22433,22439 at ./xstrdup.c:36
1210    memcpy@plt inst 22440,22448 at
1211    <unknown> inst 22441,22448 at
1212    find_charset_names inst 22455,22471 at charset.c:843,892
1213    feof@plt inst 22472,22479 at
1214    feof inst 22473,22507 at
1215    find_charset_names inst 22500,22513 at charset.c:843
1216    fgets@plt inst 22514,22514 at
1217    fgets inst 22515,22552 at
1218    __IO getline inst 22553,22554 at
1219    __IO_getline_info inst 22555,22588 at
1220    memcpy inst 22589,22608 at
1221    __IO_getline_info inst 22609,22626 at
1222    <unknown> inst 22627,22644 at
1223    __IO_getline_info inst 22645,22655 at
1224    fgets inst 22656,22682 at
1225    find_charset_names inst 22683,22687 at charset.c:851,853
1226    strlen@plt inst 22688,22688 at
1227    <unknown> inst 22689,22702 at
(gdb)
Perf infrastructure

• Provides all the necessary sideband information
  - DUMMY event to keep it coming
• Takes care of context switching
• Perf counters can be used in kernel and userspace
• Intel PT driver implements a PMU in perf
Perf infrastructure: userspace

• All the basic use cases are implemented via perf userspace
  - perf record will collect trace data
  - perf script/report will decode the traces, do all the necessary processing and translate that into perf’s “branch” and “instruction” events

• PT decoder is implemented in perf userspace
  - hard to do in real time in the kernel
  - can share data with other PT-aware tools
Example output of “perf report”

```
# Samples: 5M of event 'branches:ku'
# Event count (approx.): 5076285
#...

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.30%</td>
<td>grep</td>
<td>libc-2.15.so</td>
<td>[.] 0x00000000000027d45</td>
</tr>
<tr>
<td>16.14%</td>
<td>grep</td>
<td>libc-2.15.so</td>
<td>[.] __mbrtowc</td>
</tr>
<tr>
<td>10.48%</td>
<td>grep</td>
<td>libc-2.15.so</td>
<td>[.] wcrtomb</td>
</tr>
<tr>
<td>9.46%</td>
<td>grep</td>
<td>grep</td>
<td>[.] 0x000000000000557d</td>
</tr>
<tr>
<td>7.71%</td>
<td>grep</td>
<td>libc-2.15.so</td>
<td>[.] _dl_mcount_wrapper_check</td>
</tr>
<tr>
<td>2.11%</td>
<td>grep</td>
<td>grep</td>
<td>[.] mbrtowc@plt</td>
</tr>
<tr>
<td>1.75%</td>
<td>grep</td>
<td>libc-2.15.so</td>
<td>[.] tolower</td>
</tr>
<tr>
<td>1.75%</td>
<td>grep</td>
<td>grep</td>
<td>[.] tolower@plt</td>
</tr>
<tr>
<td>1.75%</td>
<td>grep</td>
<td>grep</td>
<td>[.] wcrtomb@plt</td>
</tr>
<tr>
<td>1.28%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] __lock_acquire</td>
</tr>
<tr>
<td>0.57%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] trace_hardirqs_off_caller</td>
</tr>
<tr>
<td>0.36%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] mark_lock</td>
</tr>
<tr>
<td>0.26%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] trace_hardirqs_off</td>
</tr>
<tr>
<td>0.25%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] lock_release</td>
</tr>
<tr>
<td>0.23%</td>
<td>grep</td>
<td>[kernel.kallsyms]</td>
<td>[.] __lock_is_held</td>
</tr>
</tbody>
</table>
```
Example output of “perf script”
Perf infrastructure: extensions

- Zero-copy mapping trace buffers to userspace
  - 8 instructions per byte of PT trace @1600MHz => 200MB/s per core
  - Can’t have kernel parse that (it’s a relatively slow and complex procedure); decoding is done by userspace
  - mmap() with a “magic” offset to map trace buffers
- Additional bits in the event attribute
- PMU capabilities
References

• Intel® Architecture Instruction Set Extensions Programming Reference

• Intel PT decoder library (BSDL)
  - https://01.org/processor-trace-decoder-library