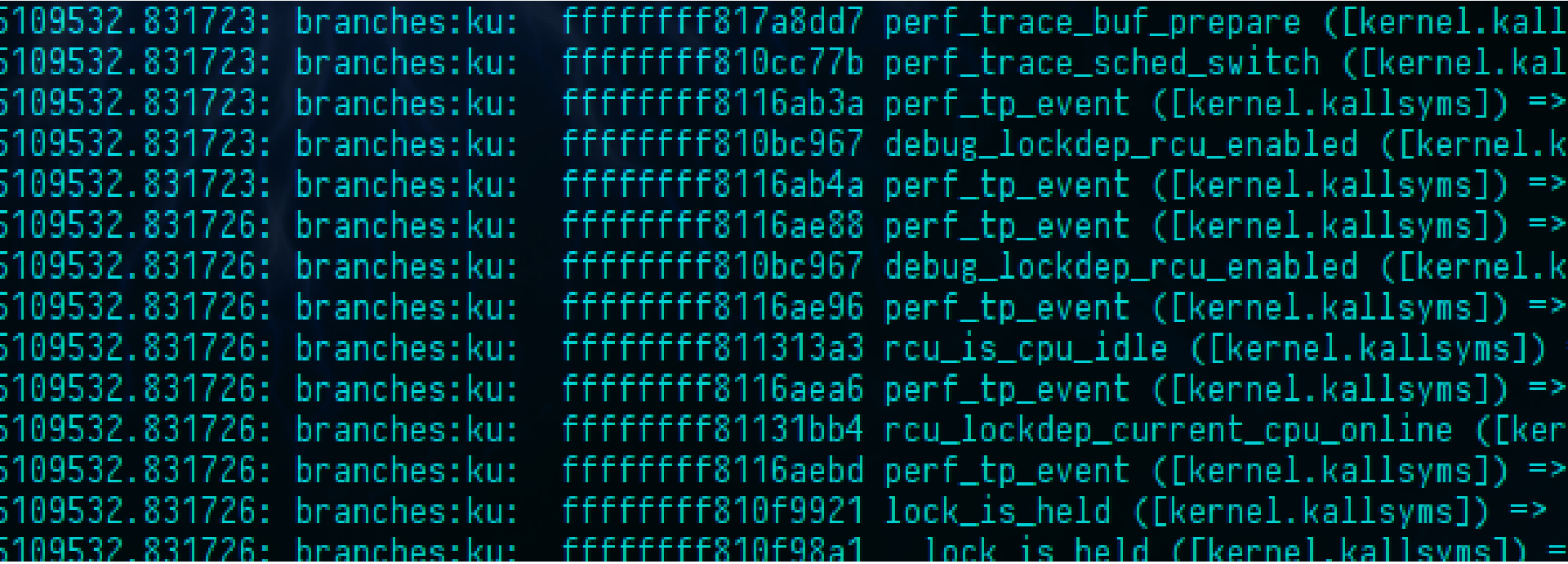
Efficient and Large Scale Program Flow Tracing in Linux

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?

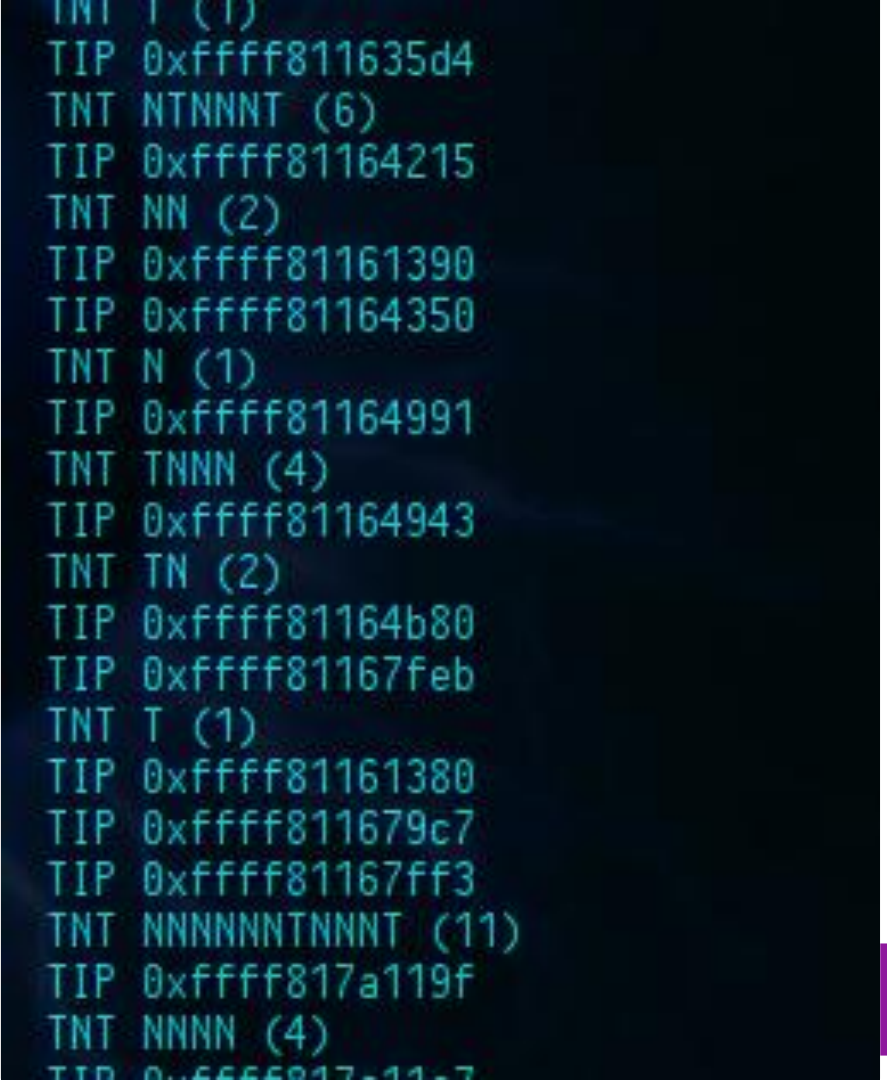
- **Branch history**
  - As opposed to stack trace
  - As opposed to function trace only (ftrace)
- **Function call graph**
- **Even better with timing information**

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


TNT T (1)
TIP 0xffff811635d4
TNT NTNNNT (6)
TIP 0xffff81164215
TNT NN (2)
TIP 0xffff81161390
TIP 0xffff81164350
TNT N (1)
TIP 0xffff81164991
TNT TNNN (4)
TIP 0xffff81164943
TNT TN (2)
TIP 0xffff81164b80
TIP 0xffff81167feb
TNT T (1)
TIP 0xffff81161380
TIP 0xffff811679c7
TIP 0xffff81167ff3
TNT NNNNNNTNNNT (11)
TIP 0xffff817a119f
TNT NNNN (4)

Intel PT

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

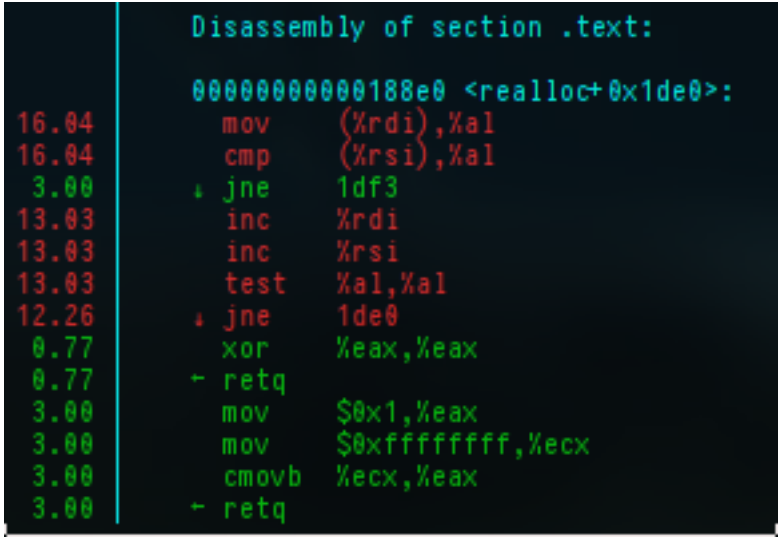
Linux support

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:52,56
1209      xstrdup       inst 22433,22439        at ./xstrdup.c:36
1210        memcpy@plt  inst 22440,22440        at
1211          <unknown> inst 22441,22454        at
1212    find_charset_names      inst 22455,22471        at charset.c:843,892
1213      feof@plt      inst 22472,22472        at
1214        feof        inst 22473,22507        at
1215    find_charset_names      inst 22508,22513        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              memchr        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
#
# Overhead  Command      Shared Object                                      Symbol
# ........  .......  ..................  ..........................................
#
    40.30%     grep  libc-2.15.so        [.] 0x0000000000027d45
    16.14%     grep  libc-2.15.so        [.] __mbrtowc
    10.48%     grep  libc-2.15.so        [.] wcrtomb
     9.46%     grep  grep                [.] 0x000000000000557d
     7.71%     grep  libc-2.15.so        [.] _dl_mcount_wrapper_check
     2.11%     grep  grep                [.] mbrtowc@plt
     1.75%     grep  libc-2.15.so        [.] towlower
     1.75%     grep  grep                [.] towlower@plt
     1.75%     grep  grep                [.] wcrtomb@plt
     1.28%     grep  [kernel.kallsyms]   [.] __lock_acquire
     0.57%     grep  [kernel.kallsyms]   [.] trace_hardirqs_off_caller
     0.36%     grep  [kernel.kallsyms]   [.] mark_lock
     0.26%     grep  [kernel.kallsyms]   [.] trace_hardirqs_off
     0.25%     grep  [kernel.kallsyms]   [.] lock_release
     0.23%     grep  [kernel.kallsyms]   [.] __lock_is_held
```


Example output of “perf script”

```
7fd5b66188ea wcrtomb (/lib/x86_64-linux-gnu/libc-2.15.so) =>           413259 [unknown] (/
      41327b [unknown] (/bin/grep) =>           402660 mbrtowc@plt (/bin/grep)
      402660 mbrtowc@plt (/bin/grep) =>     7fd5b6618570 __mbrtowc (/lib/x86_64-linux-gnu/
7fd5b661862a __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b6618639 __mbrtowc
7fd5b661863c __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b66a55a0 _dl_mcount
7fd5b66a55af _dl_mcount_wrapper_check (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b66a
7fd5b66a55d0 _dl_mcount_wrapper_check (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b661
7fd5b6618669 __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659ccf0 [unknown]
7fd5b659cd45 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659cd6c [unknown]
7fd5b659ce09 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659ce1d [unknown]
7fd5b659ce35 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659ce10 [unknown]
7fd5b659ce17 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659d0a8 [unknown]
7fd5b659d0b0 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659ce37 [unknown]
7fd5b659ce52 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659ce89 [unknown]
7fd5b659ce96 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659d78a [unknown]
7fd5b659d79a [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659cf6a [unknown]
7fd5b659cf6f [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b659cf4d [unknown]
7fd5b659cf62 [unknown] (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b661866b __mbrtowc
7fd5b6618674 __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b66186d0 __mbrtowc
7fd5b66186dd __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>     7fd5b661867f __mbrtowc
7fd5b66186cc __mbrtowc (/lib/x86_64-linux-gnu/libc-2.15.so) =>           413280 [unknown]
```


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
  - http://download-software.intel.com/sites/default/files/319433-015.pdf
- Intel PT decoder library (BSDL)
  - https://01.org/processor-trace-decoder-library