Using Perf and its friend eBPF on Arm platform
SAN19-223
Leo Yan
Linaro Support and Solutions Engineering
Introduction

This session gives an update of the latest support for Arm CoreSight and eBPF in perf in 25 minutes; if you are interested in demos then they will be presented in a CoreSight hacking session.

Firstly we will focus on Arm CoreSight branch tracing with sample flags.

In the second part, we will discuss the eBPF usage with perf tool and how perf profiles the eBPF program by using the Arm CoreSight event.

At the end, we will address integration perf for CoreSight test support.
Overview

- Perf sample flags for Arm CoreSight tracing
- Perf eBPF program support
- Perf profiles eBPF program with Arm CoreSight
- Perf test for Arm CoreSight
Perf sample flags for Arm CoreSight tracing

**perf.data for CoreSight**

- header
- ... CoreSight meta data
- ... CoreSight trace data
- ...

**perf report**

- OpenCSD

**Decoding**

**branch sample**

- ID
- PID
- end_addr
- start_addr
- flags
- insn
- insn_len
- ......
What can we know from sample flags?

Based on sample flags, we can get to know what's the purpose for the branch instruction, e.g. the branch is for function calling and returning, system call and its return, hardware interrupt and its return, etc.

It also allows to read out the instruction bytecode from DSO and shows the instruction size (A32/A64 instruction size is 32-bit, T32 instruction size can be 32-bit or 16-bit).

Sample flags mapping to Arm branch types

- jmp: Normal branch (Usually within function)
- jcc: Conditional branch
- call: Function call
- return: Normal function return
- syscall: System call
- sysret: System call return
- int: Exception (trap, fault, or alignment errors)
- hw int: Hardware Interrupt, external bus exception, debug module, PE reset and halt
- iret: Exception return
- tr strt/tr end: Trace start and end
Demo for perf sample flags

```bash
# perf record -e cs_etm/@tmc_etr0/ --per-thread -- uname
# perf script -F,+flags,+insn,+insnlen

[...]
```

Flags field displays the branch types

Instruction length and its bytecode
Next step: call chain based on sample flags

```
# perf script --itrace=g16l64i100

main  1579        100      instructions:
       ffff0000102137f0 group_sched_in+0xb0 ([kernel.kallsyms])

main  1579        100      instructions:
       ffff000010213b78 flexible_sched_in+0xf0 ([kernel.kallsyms])
       ffff00001020c0b4 visit_groups_merge+0x12c ([kernel.kallsyms])

main  1579        100      instructions:
       ffff0000102135ac event_sched_in.isra.57+0x74 ([kernel.kallsyms])
       ffff0000102137a0 group_sched_in+0x60 ([kernel.kallsyms])
       ffff000010213b84 flexible_sched_in+0xc ([kernel.kallsyms])
       ffff00001020c0b4 visit_groups_merge+0x12c ([kernel.kallsyms])

[...]
```

The sample flags can be used to create call chain: the thread stack can be pushed for the flag PERF_IP_FLAG_CALL and it will be popped for the flag PERF_IP_FLAG_RETURN, and the stack will be reset for discontinuous tracing. Finally, the thread stack can be used to create call chain.
Overview

- Perf sample flags for Arm CoreSight tracing
- Perf eBPF program support
- Perf profiles eBPF program with Arm CoreSight
- Perf test for Arm CoreSight
Using eBPF program with “perf trace” tool

“perf trace” tool allows to specify eBPF program as an event with option \texttt{-e}. We firstly need to define the environment variables in \texttt{~/.perfconfig}, this allows perf tool to have enough knowledge for eBPF compilation.

The tool will automatically invoke LLVM/Clang to build the eBPF bytecode and load the program into kernel; perf will automatically bind the event with eBPF program.

```
# cat ~/.perfconfig
[llvm]
clang-path = /bin/clang-8
kbuild-dir = $linux-kernel
clang-opt = "-g"
dump-obj = true
```

```
# perf trace -e examples/bpf/hello.c
LLVM: dumping examples/bpf/hello.o
```
Extending eBPF loader in perf

```c
#include <bpf.h>

int probe(hrtimer_nanosleep, rqtp->tv_sec) {
    (void *)ctx, int err, long sec)
    {
        return sec == 5;
    }

license(GPL);
```

When perf tool loads eBPF program, it can extend the eBPF program's prologue and use this chance to insert the specific instructions for calling eBPF functions, e.g. `bpf_probe_read()` or `bpf_probe_read_string()`, this allows the program to access data structure element more easily.
Using perf to trace system call [1]

strace is well known for system call trace, since it relies on ptrace to pause the application to gather system call related information, this causes very poor performance when it traces system call.

perf trace uses trace points to collect system call info, thus the workload doesn’t need to stop for tracing.

Furthermore, to support more complex argument formats (e.g. strings, array, etc) rather than only integer argument, perf trace extends to use eBPF program for syscall argument tracing.

Performance comparison for system call tracing (on Juno)

```
# dd if=/dev/zero of=/dev/null bs=1 count=5000k
5120000+0 records in
5120000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 6.53434 s, 784 kB/s

# strace -e accept \ 
   dd if=/dev/zero of=/dev/null bs=1 count=5000k
5120000+0 records in
5120000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 232.335 s, 22.0 kB/s
+++ exited with 0 +++

# perf trace -e accept -e augmented_raw_syscalls.c -- \ 
   dd if=/dev/zero of=/dev/null bs=1 count=5000k
5120000+0 records in
5120000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 16.447 s, 311 kB/s
```
```c
int sys_enter(struct syscall_enter_args *args)
{
    struct augmented_args_payload *augmented_args = augmented_args_payload();
    const void *filename_arg = (const void *)args->args[1];
    unsigned int len = sizeof(augmented_args->args);

    if (augmented_args == NULL)
        return 1; /* Failure: don't filter */

    if (syscalls_sys_enter_nr != augmented_args->syscall_nr)
        return 0;

    bpf_tail_call(args, &syscalls_sys_enter, augmented_args->args.syscall_nr);
    return 0;
}
```

A bpf map is used to maintain the mapping between syscall # and augment handling prog; so \textbf{bpf\_tail\_call()} will invoke the corresponding program with the specified syscall number.
Demo for perf trace to trace system call info

```shell
# perf trace -e string \
   -e $linux/tools/perf/examples/bpf/augmented_raw_syscalls.c

[...,]

vi/16903 openat(AT_FDCWD, "/root/.viminfo", O_RDONLY) = 3
vi/16903 openat(AT_FDCWD, "/root/.viminfo.tmp", O_CREAT|O_EXCL|O_LARGEFILE|O_WRONLY, S_IRUSR|S_IWUSR) = 7
vi/16903 unlinkat(AT_FDCWD, "/root/.viminfo", 0) = 0
vi/16903 renameat(AT_FDCWD, "/root/.viminfo.tmp", AT_FDCWD, "/root/.viminfo") = 0
cscope/16904 unlinkat(AT_FDCWD, "/tmp/cscope.16904/cscope.1", 0) = -1 ENOENT (No such file or directory)
cscope/16904 unlinkat(AT_FDCWD, "/tmp/cscope.16904/cscope.2", 0) = -1 ENOENT (No such file or directory)
cscope/16904 unlinkat(AT_FDCWD, "/tmp/cscope.16904", 512) = 0

[...]```
Overview

- Perf sample flags for Arm CoreSight tracing
- Perf eBPF program support
- Perf profiles eBPF program with Arm CoreSight
- Perf test for Arm CoreSight
Can eBPF program be profiled by perf?

Though eBPF program is light weight tracing tool with its filtering capability, it still might introduce overhead for system, especially if trace the hot path; e.g. in LPC2019, Arnaldo Carvalho de Melo’s speech “BPF is eating the world, don't you see?” \[^{[2]}\] presented examples for heavy overhead with eBPF programs.

If without profiling eBPF program, perf will miss the eBPF part; thus it might output incomplete workloads statistics for the whole system; this might introduce bias for the system which uses eBPF filter functionality heavily.

BTF (BPF type format) is introduced to encode debug infos related to BPF program/map; except it defines the ELF file format, it also defines the API between the kernel and user space, thus it gives perf chance to read out the debug info and annotate the code.

pahole tool \[^{[3]}\] and llvm 8.0 or later version support BTF format, so for perf profiling, we need use them to generate BTF for eBPF program ahead.
Issue: eBPF is missed profiling with CoreSight

The memory regions which has less address than '_stext' will not be considered as the kernel address, thus the related tracing data will be missed to find its correct DSO and fail to generate samples for eBPF program. To fix this issue, we should traverse the kallsyms which contains the exported eBPF program symbols and dynamically fixup 'machine->kernel_start'.
Demo for profiling eBPF with Arm CoreSight

**Step 1: Configure ~/.perfconfig**

root@debian:~# cat ~/.perfconfig
# this file is auto-generated by
# 'perf config'.

[llvm]
clang-path = /bin/clang-8
kbuild-dir = $linux-kernel
clang-opt = "-g"
dump-obj = true

[trace]
show_zeros = yes
show_duration = no
no_inherit = yes
show_timestamp = no
show_arg_names = no
args_alignment = 40
show_prefix = yes

**Step 2: Run 'perf trace' command with eBPF event**

# perf trace -e string -e augmented_raw_syscalls.c

**Step 3: Read eBPF program memory mapping in kernel**

# echo 1 > /proc/sys/net/core/bpf_jit_kallsyms
# cat /proc/kallsyms | grep -E "bpf_prog_.+_sys_[enter|exit]"

ffff000000086a84 t bpf_prog_f173133dc38ccf87_sys_enter [bpf]
ffff000000088618 t bpf_prog_c1bd85c092d6e4aa_sys_exit [bpf]
Demo for profiling eBPF with Arm CoreSight - cont.

**Step 4: Capture CoreSight trace data with eBPF program**

```
# perf record -e cs_etm/@20070000.etr/ \ 
   --filter 'filter 0xffff000000086a84/0x800' \ 
   -a sleep 5s
```

**Step 5: Annotate eBPF program**

```
# perf report
Then select 'branches' samples and press 'a' to annotate 'bpf_prog_f173133dc38ccf87_sys_enter', press 'P' to print to the .annotation file.
```

```c
bpf_prog_f173133dc38ccf87_sys_enter.annotation

bpf_prog_f173133dc38ccf87_sys_enter()

Event: branches

Percent  int sys_enter(struct syscall_enter_args *args)
    stp  x29, x30, [sp, #-16]!

    int key = 0;
    mov  x29, sp

    augmented_args =
        bpf_map_lookup_elem(&augmented_filename_map, &key);  
    stp  x19, x20, [sp, #-16]!

    augmented_args =
        bpf_map_lookup_elem(&augmented_filename_map, &key);

    [...]
```
Overview

- Perf sample flags for Arm CoreSight tracing
- Perf eBPF program support
- Perf profiles eBPF program with Arm CoreSight
- Perf test for Arm CoreSight
Avoiding regressions in Coresight framework

We need a ‘standard’ test to verify incoming CoreSight driver and perf related patches for the mainline kernel, so it can allow all developers to use the same case for smoke testing.

The test should not be too complex but it needs to cover the main functionalities verification.

It’s good to utilize the existed test framework and should be easily fetched by developers for using it (so ideally the test code should be merged in the mainline kernel).

The CoreSight test case bases on ‘perf test’ tool so can utilize the perf test framework for the testing. Furthermore, it also verifies the trace data decoding with ‘perf script’ and ‘perf report’ commands.
Challenge: multiple data paths

Path 1: Core0 -> ETF
Path 2: Core0 -> ETF -> ETR
Path 3: Core1 -> ETF
Path 4: Core1 -> ETF -> ETR
Depth first search (DFS) for traversing data paths

From every ETM’s point of view, it's a root of the tree for the connection with links and sinks. Since there have multiple ETM devices, we use depth first search algorithm to traverse all possible paths for every ETM and find combinations between the ETM and sinks.

As result, the testing sequence is as below for right diagram:

ETM0 -> ETF
ETM0 -> ETR
ETM1 -> ETF
ETM1 -> ETR

```bash
perf record -e cs_etm/@tmc_et{f|r}0/ --per-thread -- taskset -c {0|1} touch file
```
Demo for perf test

```bash
# cd $linux/tools/perf  -> This is important so can use shell script
# ./perf test list

[...

61: Check Arm CoreSight trace data recording and branch samples
62: Check open filename arg using perf trace + vfs_getname
63: Zstd perf.data compression/decompression
64: Add vfs_getname probe to get syscall args filenames

# ./perf test 61

61: Check Arm CoreSight trace data recording and branch samples: Ok
```

Find the test ID for CoreSight.

Specify the test ID to execute CoreSight testing.
Further reading

[1] perf lab - Tips, tricks and a DB optimization using perf
   - Arnaldo Carvalho de Melo, Jiří Olša, Joe Mario (Redhat)
[2] pahole tool: git repository
[3] BPF is eating the world, don't you see? - Arnaldo Carvalho de Melo
[4] Perf sample flags for Arm CoreSight tracing (merged in mainline)
   - Sample flags patch v7  Sample flags 'insn' and 'insnlen'
[5] Perf callchain support for Arm CoreSight tracing (upstreaming)
   - perf cs-etm: Support thread stack and callchain
[6] Perf eBPF program support on Arm64 (merged in mainline)
   - Arnaldo's patches: [1] [2]
   - perf trace: Exit when failing to build eBPF program
[7] Perf profiles eBPF program with Arm CoreSight (upstreaming)
   - perf machine: arm/arm64: Improve completeness for kernel address space
[8] Perf test for Arm CoreSight (pending for sysfs node refactoring)
   - perf test: Introduce script for Arm CoreSight testing
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org