Linaro connect
Las Vegas 2016

利用CoreSight 和 OpenCSD在ARM上做硬件辅助tracing

(张春艳 译自 Mathieu Poirier)

ENGINEERS
AND DEVICES
WORKING
TOGETHER

此次报告将覆盖到:

- 端到端的技术概述
- 并不在细节上深入介绍CoreSight
- 重点在于如何使用它而不是它是什么
- 主要会讲到CoreSight和Linux Perf集成
- 将CoreSight初步用起来需要的一切

- 基于以上
  - 一个简短的CoreSight介绍
  - 如何在系统中使能CoreSight
  - 用于解码跟踪数据的开源库OpenCSD
  - 如何获取跟踪数据
  - 解码原始跟踪数据

CoreSight是什么

- 多项系统跟踪技术的统称
- 包括在SoC上做系统追踪所需的一切东东, 可独立工作也可外接辅助工具

- 我们的工作主要聚焦在硬件辅助追踪和解析这些追踪数据

- 什么是硬件辅助追踪?
  - 追踪CPU core做了什么, 而且**不影响**CPU性能
  - 不用外接其他硬件
  - CPU core并非必须运行Linux!

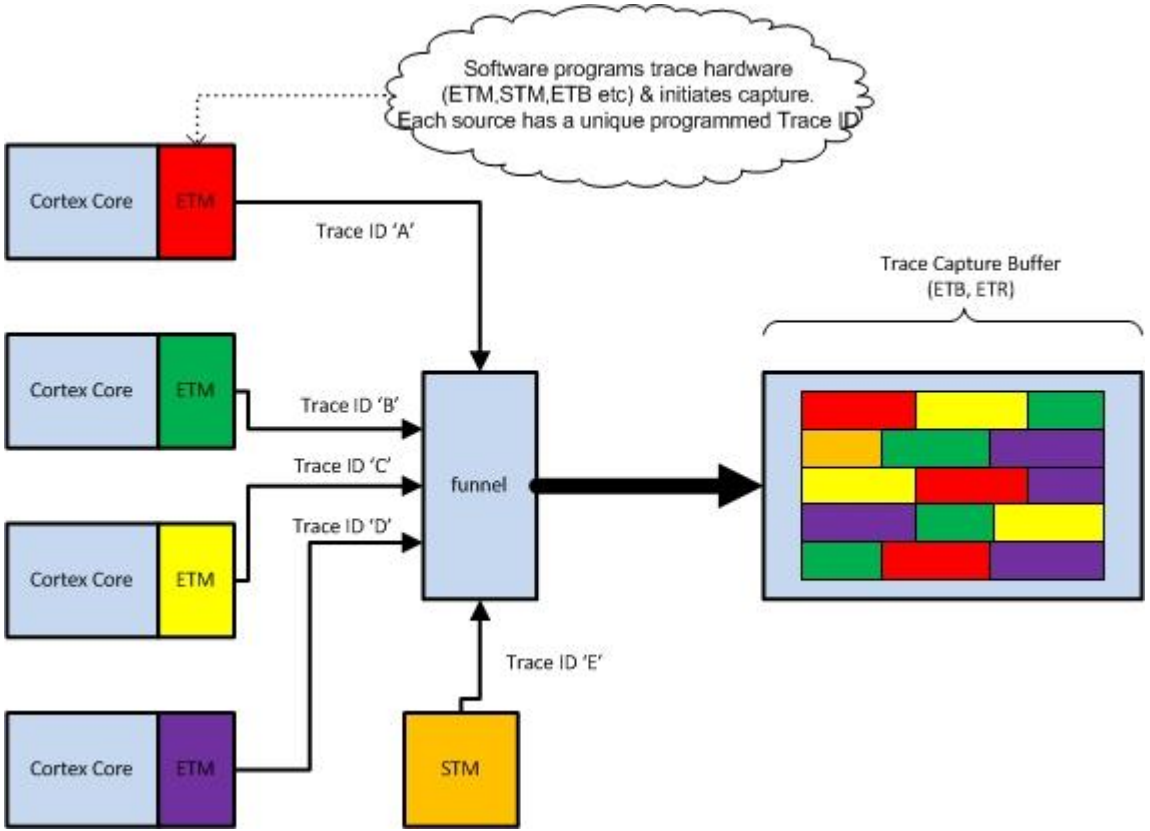
- CoreSight驱动及其基础结构的源代码在Kernel如下目录中

```
drivers/hwtracing/coresight/
```

硬件辅助Tracing如何工作?

- 系统中的每个CPU core都被配置了相应的一个IP核 - **Embedded Trace Macrocel (ETM)**
- 典型情况是每个CPU core对应一个Embedded Trace Macrocell
- 系统驱动已对trace macrocell很多跟踪特性做好了配置
  - 在接下来的幻灯片中有很多使用案例

- 被触发追踪动作之后, trace macrocells的操作都是独立
- 不牵扯CPU core, 所以对CPU性能没有影响
  - 这里要注意下CoreSight拓扑结构和内存总线布局


Software programs trace hardware
(ETM,STM,ETB etc) & initiates capture.
Each source has a unique programmed Trace ID
Cortex Core
ETM
Trace ID 'A'
Cortex Core
ETM
Trace ID 'B'
Trace ID 'C'
Cortex Core
ETM
Trace ID 'D'
Cortex Core
ETM
funnel
Trace Capture Buffer
(ETB, ETR)
Trace ID 'E'
STM

Program Flow Trace

- 由硬件产生的一种格式为**Program flow trace (PFT)** 的追踪数据
- **Program flow trace**是一系列(程序执行过程中)被处理器踩过的**Waypoints**
- **Waypoints** 包括:
  - 某些分支指令 (branch instruction)
  - 异常 (Exceptions)
  - 返回指令 (Returns)
  - 内存屏障 (Memory barriers)

- 利用程序镜像和那些Waypoints, 就可以重构处理器运行过的代码路径.
- 可以用OpenCSD将**Program flow traces**解码成指令区间

CoreSight On Your System

- 所有CoreSight 组件的代码都已经合到内核主线了
- 除了CTI 和 ITM的驱动代码还没有
  - CTI 很快就可用了
  - ITM 是一个比较老的IP - 相对较容易支持

- 参考平台有 Vexpress (ARMv7) 和 Juno (ARMv8)

- 对于任何平台, CoreSight的拓扑结构都隐藏在DT中
- 拓扑结构的描述采用 V4L2 graph bindings形式
  - 参考平台的DT文件都已经合入Linux主线了, 几乎可以覆盖所有案例
  - http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/graph.txt

- 只要DT配置是对的, CoreSight应该就可以工作了...

CoreSight - 共有的难点

- 有很多方面要处理
  - 像其它功能强大的技术一样, CoreSight也是很复杂的
  - 将CoreSight集成到Perf架构中, 解决了一多半比较难搞的问题
  - OpenCSD解码库搞定了剩下的难题

- 电源域和时钟问题:
  - 在多数的硬件方案中, CoreSight各子模块, 一部分在CPU core的电源域, 另一部分在debug电源域.
  - CoreSight时钟要使能 → 驱动程序应该会处理 (只要DT写的是对的)

- 电源域管理:
  - Trace macrocells一般是和与之关联的CPU共享同一个电源域
  - 如果CPUidle将CPU置于深度睡眠状态, 电源域通常就被下电了
  - ***** 不要在使能了CPUidle的情况下使用CoreSight *****
  - 做自己的解决方案时, 记得考虑 “PowerDown control” 寄存器 (TRCPDCR:PU)!

Booting with CoreSight Enabled

```
sdhci-pltfm: SDHCI platform and OF driver helper
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
coresight-etm4x 22040000.etm: ETM 4.0 initialized
coresight-etm4x 22140000.etm: ETM 4.0 initialized
coresight-etm4x 23040000.etm: ETM 4.0 initialized
coresight-etm4x 23140000.etm: ETM 4.0 initialized
coresight-etm4x 23240000.etm: ETM 4.0 initialized
coresight-etm4x 23340000.etm: ETM 4.0 initialized
usb 1-1: new high-speed USB device number 2 using ehci-platform
NET: Registered protocol family 17
9pnet: Installing 9P2000 support
```

```
root@linaro-nano:~# ls /sys/bus/coresight/devices/
20010000.etf              220c0000.cluster0-funnel  23240000.etm
20030000.tpiu             22140000.etm              23340000.etm
20040000.main-funnel      23040000.etm              coresight-replicator
20070000.etr              230c0000.cluster1-funnel
22040000.etm              23140000.etm
root@linaro-nano:~#
```

CoreSight 和 Perf集成

- Perf 可谓无处不在, 有很好的说明文档, 也被Linux开发者们大量地使用
- Perf 有一套专为系统追踪而设计的框架
- 将大部分CoreSight固有的复杂性封装起来对用户隐藏
- 提供了便于将系统追踪的解码功能集成的工具
  - 不用处理那个 “metadata”

- Trace Macrocell 对于Perf core来讲就是PMU (Performance Monitor Unit)
  - 非常紧密的控制跟踪动作何时开始和结束
  - 绘制跟踪数据时可以在用户空间和内核空间之间做到零拷贝

- 注册PMU是在CoreSight公共框架中完成的 → (不需要介入)
- CoreSight PMU在Perf core中的名字是cs_etm.

CoreSight 的各跟踪器作为PMU如下所示

```
linaro@linaro-nano:~$ tree /sys/bus/event_source/devices/cs_etm
/sys/bus/event_source/devices/cs_etm
├── cpu0 -> ../platform/23040000.etm/23040000.etm
├── cpu1 -> ../platform/22040000.etm/22040000.etm
├── cpu2 -> ../platform/22140000.etm/22140000.etm
├── cpu3 -> ../platform/23140000.etm/23140000.etm
├── cpu4 -> ../platform/23240000.etm/23240000.etm
├── cpu5 -> ../platform/23340000.etm/23340000.etm
├── format
│   ├── cycacc
│   └── timestamp
├── nr_addr_filters
├── perf_event_mux_interval_ms
├── power
│   ├── autosuspend_delay_ms
│   ├── control
│   ├── runtime_active_time
│   ├── runtime_status
│   └── runtime_suspended_time
├── subsystem -> ../../bus/event_source
├── type
└── uevent
9 directories, 11 files
linaro@linaro-nano:~$
```

普通的 sysFS PMU 接口

OpenCSD 用于追踪数据解码

- Open **C**ore**S**ight **D**ecoding library (开源CoreSight解码库)
- 一项由Texas Instrument, ARM and Linaro联合开发的工作
- 免费且开源的Program Flow Traces的解码方案
- 目前支持 ETMv3, PTM 和 ETMv4解码
- 也支持 MIPI trace 解码 (即输出自STM的追踪数据)
- 完全和Perf集成了
- 任何人都可以到gitHub**[1]**下载源码, 做集成或者修改等
- 一篇对此做了深入讲解的文章不久前被发表在CoreDump blog **[2]**

[1]. https://github.com/Linaro/OpenCSD

[2]. http://www.linaro.org/blog/core-dump/opencsd-operation-use-library/

把上面讲的放在一起

现在我们已经知道了....

- 我们可以在ARM平台上利用CoreSight IP blocks做硬件辅助追踪
- Linux 内核提供了一套架构和一系列驱动支持CoreSight
- openCSD 解码库也已经做好了, 任何人都可以取用解码CoreSight的追踪数据
- CoreSight 和 openCSD 已经和Perf子系统集成好了

- 是时候看看如何把这些东西放在一起, 并在真实的场景中用起来

获取正确的工具

- 首先, 需要下载OpenCSD解码库
  - gitHub[1] 的 master分支上有OpenCSD源码
  - 稳定版都打着tag
  - 旧版本有专属分支 -- 请跟紧最新版本
  - 文件"HOWTO.md" 会告诉你哪个Kernel版本和最新版的OpenCSD是匹配的
  - 其上的Kernel分支会在不久的将来消失 - 那时所有在Linux tree上的开发都合入Linux主线了

- gitHub上的Kernel分支包含着用户空间的功能
  - 总是有一个以最新版的 Kernel为基线的版本
  - perf [record, report, script]
  - 把这些工具提交进内核主线的工作正在进行中
  - 如果要用CoreSight和Perf集成的功能, 请在自己的定制tree上包含这些补丁

[1]. https://github.com/Linaro/OpenCSD

编译 OpenCSD 和 Perf 工具

- OpenCSD 是一个独立的库 - 所以它不是kernel tree的一部分
- OpenCSD 库需要与Perf工具链接在一起
  - 如果Perf和OpenCSD没有联在一起用, 将做不了追踪数据的解 码

- 参考 github上的文档**"HOWTO.md"** 的指导
- 要设置环境变量 "CSTRACE_PATH"以支持对追数据的解码功能

```
CC       tests/thread-mg-share.o
CC       util/cs-etm-decoder/cs-etm-decoder-stub.o
CC       util/intel-pt-decoder/intel-pt-decoder.o
```

不带CS解码功能

```
CC       util/auxtrace.o
CC       util/cs-etm-decoder/cs-etm-decoder.o
LD       util/cs-etm-decoder/libperf-in.o
```

带CS解码功能

通过Perf工具使用CoreSight

- CoreSight PMU和其它的PMU用法一样

```
./perf record -e event_name/{options}/ --perf-thread ./main
```

- 所以, 最简单的命令格式如下:

```
./perf record -e cs_etm/@20070000.etr/ --perf-thread ./main
```

- 每次都要指定一个sink设备来表明要把跟踪数据输出到哪里
  - CoreSight 子设备列表在sysfs下可查看到

```
linaro@linaro-nano:~$ ls /sys/bus/coresight/devices/
20010000.etf              20040000.main-funnel      22040000.etm              22140000.etm
230c0000.cluster1-funnel  23240000.etm             coresight-replicator      20030000.tpiu
20070000.etr              220c0000.cluster0-funnel  23040000.etm              23140000.etm
23340000.etm
```

通过Perf工具使用CoreSight (接续)

- 默认的选项会产生过多的跟踪数据
- 选项 'k' 和 'u' 分别用于限制Perf仅追踪内核(kernel)空间和用户(user)空间

```
./perf record -e cs_etm/@20070000.etr/u --perf-thread ./main
./perf record -e cs_etm/@20070000.etr/k --perf-thread ./main
```

- 追踪内核空间需要 root 权限

- (指令)地址过滤器可以用来限制仅追踪指定的区域
  - "地址区间"过滤器 → 用 "filter" 关键字
  - "开始/停止"过滤器→ 用 "start" 和 "stop" 关键字

使用CoreSight “地址区间”过滤器

- 跟踪两个地址之间的指令
- **不包括**跳转到地址区间之外执行的那些指令

内核空间示例:

```
$ perf record -e cs_etm/@20010000.etr/k --filter            \
                        'filter 0xffffff8008562d0c/0x48' --per-thread ./main
```

用户空间示例:

```
$ perf record -e cs_etm/@20070000.etr/u --filter            \
                        'filter 0x72c/0x40@/opt/lib/libcstest.so.1.0' --per-thread ./main
```

使用 CoreSight “开始/停止”过滤器

- 在一个指令地址处开始, 在另一个结束
- **包括**跳转到地址区间之外执行的那些指令

内核空间示例:

```
perf record -e cs_etm/@20070000.etr/k --filter                                  \
                    'start 0xffffff800856bc50,stop 0xffffff800856bcb0' --per-thread ./main

perf record -e cs_etm/@20070000.etr/k --filter                                  \
                    'start 0xffffff800856bc50,stop 0xffffff800856bcb0,          \
                     start 0xffffff8008562d0c,stop 0xffffff8008562d30' --per-thread ./main
```

用户空间示例:

```
perf record -e cs_etm/@20070000.etr/u --filter                              \
                         'start 0x72c@/opt/lib/libcstest.so.1.0,            \
                         stop 0x26@/main' --per-thread ./main
```

CoreSight 过滤器的限制

- Trace Macrocells的地址比较器数量是有限的
  - 依赖于硬件实现, 目前的限制是最多 8对

- “地址区间” 和 “开始/停止” 过滤器不能在同一个追踪任务中一起用

下面这个示例是**不支持**的:

```
perf record -e cs_etm/@20070000.etr/k --filter                                  \
                     'start 0xffffff800856bc50,stop 0xffffff800856bcb0',    \   // start/stop
                      filter 0x72c/0x40@/opt/lib/libcstest.so.1.0'          \   // Range
                     --per-thread ./main
```

通过Perf工具使用CoreSight (接续)

- 追踪数据保存在 “perf.data” 文件中

  `perf report --dump perf.data`

```
0x728 [0x30]: PERF_RECORD_AUXTRACE size: 0xf0  offset: 0  ref: 0x48b2b5695d22eed5  idx: 0  tid: 1796
cpu: -1

. ... CoreSight ETM Trace data: size 240 bytes
  0: I_ASYNC : Alignment Synchronisation.
  12: I_TRACE_INFO : Trace Info.
  17: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  48: I_ASYNC : Alignment Synchronisation.
  60: I_TRACE_INFO : Trace Info.
  65: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  96: I_ASYNC : Alignment Synchronisation.
  108: I_TRACE_INFO : Trace Info.
  113: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  144: I_ASYNC : Alignment Synchronisation.
  ....
```

一个简单但真实的示例

“main.c”

```
#include <stdio.h>

int coresight_test1(int val);

int main(void)
{
     int val;

     val = coresight_test1(10);

     printf("val: %d\n", val);

     return 0;
}
```

“libcstest.c”

```
int coresight_test1(int val)
{
     int i;
     /*
      * A simple loop forcing the
      * instruction pointer to move
      * around.
      */
     for (i = 0; i < 5; i++)
          val += 2;

          return val;
     }
```

Code to trace

Objdump the Code to Trace

```
$ aarch64-linux-gnu-objdump -d libcstest.so.1.0

000000000000072c <coresight_test1>:
 72c:   d10083ff        sub     sp, sp, #0x20
 730:   b9000fe0        str     w0, [sp,#12]
 734:   b9001fff        str     wzr, [sp,#28]
 738:   14000007        b       754 <coresight_test1+0x28>
 73c:   b9400fe0        ldr     w0, [sp,#12]
 740:   11000800        add     w0, w0, #0x2
 744:   b9000fe0        str     w0, [sp,#12]
 748:   b9401fe0        ldr     w0, [sp,#28]
 74c:   11000400        add     w0, w0, #0x1
 750:   b9001fe0        str     w0, [sp,#28]
 754:   b9401fe0        ldr     w0, [sp,#28]
 758:   7100101f        cmp     w0, #0x4
 75c:   54ffff0d        b.le    73c <coresight_test1+0x10>
 760:   b9400fe0        ldr     w0, [sp,#12]
 764:   910083ff        add     sp, sp, #0x20
 768:   d65f03c0        ret
```

在目标平台上产生追踪数据

```
root@linaro-nano:~# date
Wed Sep  7 20:17:36 UTC 2016

root@linaro-nano:~# uname -mr
4.8.0-rc5+ aarch64
root@linaro-nano:~# ls /opt/lib/libcstest.so*
/opt/lib/libcstest.so  /opt/lib/libcstest.so.1  /opt/lib/libcstest.so.1.0

root@linaro-nano:~# rm -rf ~/.debug
root@linaro-nano:~# echo 0 > /proc/sys/kernel/kptr_restrict
root@linaro-nano:~# perf record -e cs_etm/@20070000.etr/u --filter 'filter  \
                    0x72c/0x40@/opt/lib/libcstest.so.1.0' --per-thread ./main
val: 20
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.002 MB perf.data ]

root@linaro-nano:~# ls -l perf.data
-rw------- 1 root root 8176 Sep  7 20:17 perf.data
```

在目标平台上收集追踪数据

```
root@linaro-nano:~# ls -l perf.data
-rw------- 1 root root 8176 Sep  7 20:17 perf.data

root@linaro-nano:~# tar czf cs_example.tgz perf.data ~/.debug
```

- 为什么我们需要~/.debug 目录?
  - 因为它包含着追踪任 务中所涉及到的二 进制文件的一个快照
  - Perf自带的
  - 替你收集所有的一切 - 除了内核镜像

".debug" 目录的重要性

```
root@linaro-nano:~# tree .debug
.debug
├── [kernel.kallsyms]
│   └── 942a60ae69427f5dbaa1c3541671e504509bd5db
│       └── kallsyms
├── [vdso]
│   └── f1e1d7c7f2c709fb14ee135018417767eecbc0dd
│       └── vdso
├── home
│   └── linaro
│       └── main
│           └── 9a6850fab2ebbe386d3619bce3674a55622f2872
│               └── elf
├── lib
│   └── aarch64-linux-gnu
│       ├── ld-2.21.so
│       │   └── 94912dc5a1dc8c7ef2c4e4649d4b1639b6ebc8b7
│       │       └── elf
│       └── libc-2.21.so
│           └── 169a143e9c40cfd9d09695333e45fd67743cd2d6
│               └── elf
└── opt
    └── lib
        └── libcstest.so.1.0
            └── 3b3051b8a67f212a66e383fc90db3c2bde8f936f
                └── elf

18 directories, 6 files
```

....

离线的追踪数据解码: “perf report”

```
$ tar xf cs_example.tgz
$ rm -rf ~/.debug                // remove previous trace data
$ cp -dpR .debug ~/              // copy the current trace data
$ perf report --stdio            // by default file “perf.data” is used

# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#

# Samples: 8  of event 'instructions:u'
# Event count (approx.): 55
#
# Children      Self  Command  Shared Object     Symbol
# ........  ........  .......  ................  ......................
#
    81.82%    81.82%  main     libcstest.so.1.0  [.] 0x000000000000073c
     7.27%     7.27%  main     libcstest.so.1.0  [.] 0x000000000000072c
     5.45%     5.45%  main     libcstest.so.1.0  [.] 0x0000000000000754
     5.45%     5.45%  main     libcstest.so.1.0  [.] 0x0000000000000760
```

离线的追踪数据解码: “perf script”

```
$ perf script
     main  1796          4   instructions:u:       7fb19c972c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          3   instructions:u:       7fb19c9754 [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:       7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:       7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:       7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:       7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:       7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          3   instructions:u:       7fb19c9760 [unknown] (/opt/lib/libcstest.so.1.0)
```

VMA portion

ELF portion

离线的追踪数据解码: “perf script”

```
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c972c:d10083ff     sub    sp, sp, #0x20
        7fb19c9730:b9000fe0     str    w0, [sp,#12]
        7fb19c9734:b9001fff     str    wzr, [sp,#28]
        7fb19c9738:14000007     b      7fb19c9754 <__gmon_start__@plt+0x134>
```

- 地址中第一部分是哪里来的?

```
$ perf script --show-mmap-events | grep PERF_RECORD_MMAP2
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x400000(0x1000) @ 0 08:02 33169 1522333852]: r-xp /home/linaro/main
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb19db000(0x2f000) @ 0 08:02 574 1811179601]: r-xp
/lib/aarch64-linux-gnu/ld-2.21.so
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb19c9000(0x12000) @ 0 08:02 38308 4289568329]: r-xp
/opt/lib/libcstest.so.1.0
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb1880000(0x149000) @ 0 08:02 543 1811179570]: r-xp
/lib/aarch64-linux-gnu/libc-2.21.so
```

离线的追踪数据解码: “perf script”

```
$ cat range.sh
#!/bin/bash

EXEC_PATH=${HOME}/work/linaro/coresight/kernel-stm/tools/perf/
SCRIPT_PATH=${EXEC_PATH}/scripts/python/
XTOOLS_PATH=${HOME}/work/linaro/coresight/toolchain/gcc-linaro-aarch64-linux-gnu-4.8-2013.11_linux/bin/

perf --exec-path=${EXEC_PATH} script --script=python:${SCRIPT_PATH}/cs-trace-ranges.py

$ ./range.sh
range: 7fb19c972c - 7fb19c973c
range: 7fb19c9754 - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c9760 - 7fb19c976c
```

离线的追踪数据解码: “perf script”

```
$ cat disasm.py
#!/bin/bash

EXEC_PATH=${HOME}/work/linaro/coresight/kernel-stm/tools/perf/
SCRIPT_PATH=${EXEC_PATH}/scripts/python/
XTOOLS_PATH=${HOME}/work/linaro/coresight/toolchain/gcc-linaro-aarch64-linux-gnu-4.8-2013.11_linux/bin/

perf --exec-path=${EXEC_PATH}                                                    \
                          script --script=python:${SCRIPT_PATH}/cs-trace-disasm.py --    \
                          -d ${XTOOLS_PATH}/aarch64-linux-gnu-objdump
```

离线的追踪数据解码: “perf script”

```
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c972c:d10083ff     sub    sp, sp, #0x20
        7fb19c9730:b9000fe0     str    w0, [sp,#12]
        7fb19c9734:b9001fff     str    wzr, [sp,#28]
        7fb19c9738:14000007     b      7fb19c9754 <__gmon_start__@plt+0x134>
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c9754:b9401fe0     ldr    w0, [sp,#28]
        7fb19c9758:7100101f     cmp    w0, #0x4
        7fb19c975c:54ffff0d     b.le   7fb19c973c <__gmon_start__@plt+0x11c>
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c973c:b9400fe0     ldr    w0, [sp,#12]
        7fb19c9740:11000800     add    w0, w0, #0x2
        7fb19c9744:b9000fe0     str    w0, [sp,#12]
        7fb19c9748:b9401fe0     ldr    w0, [sp,#28]
        7fb19c974c:11000400     add    w0, w0, #0x1
        7fb19c9750:b9001fe0     str    w0, [sp,#28]
        7fb19c9754:b9401fe0     ldr    w0, [sp,#28]
        7fb19c9758:7100101f     cmp    w0, #0x4
        7fb19c975c:54ffff0d     b.le   7fb19c973c <__gmon_start__@plt+0x11c>
...
...
```

还没提到的几件事

- 和Perf的集成部分:
  - 在内核空间收集追踪数据时, 文件“vmlinux” 不会被包含在 .debug 目录中
  - 支持 “snapshot mode” 可以让用户抓到无穷无尽的追踪数据
  - 目前只做了ARMv8 CoreSight和 perf的集成 → 在ARMv7上做这个会比较简单的
- 有意地让在CoreSight上进行的Perf操作和在Intel PT上一样
- CoreSight 公共架构和驱动也可以通过sysFS来使用
- Upstreaming
  - 这套方案中所有内核空间的代码将会合入内核主线4.9
  - 对于用户空间, 亦即 “perf tools” 正在积极地向社区提交
- 很快会支持“Cross Trigger Interface” (CTI)

谢谢参与

The Linaro CoreSight Team:

Chunyan Zhang

Tor Jeremiassen

Mike Leach

Serge Broslavsky

Mathieu Poirier

Thank You

#LAS16

For further information: www.linaro.org

LAS16 keynotes and videos on: connect.linaro.org