Postmortem Debugging with Coresight
HKG18-TR14

Leo Yan, Linaro Support and Solutions Engineering
Introduction

This session discusses postmortem debugging techniques in the Linux kernel.

Firstly we will review ramoops (aka. pstore) with software tracing for postmortem debugging, then go through two Coresight debugging methods with hardware assisted tracing.

We will finish this material + 3 demos in 25 minutes.
Overview

● Discussion for practical scenarios
  ○ Finding program execution flow for system hang
  ○ CPU is dead, how to read current CPU state?
  ○ Offline analysis of program execution flow

● You can extend for your SoC
The scenario - Finding program execution flow for system hang

When a system hangs, we cannot always rely on the console for printing log messages.

Modern CPUs improve performance by using asynchronous operations on external bus or memory system (early acknowledge, OoO, etc), such systems may not hang immediately after running a ‘bad’ instruction.

To narrow down the cause, we need to reverse program execution flow to find hints.

- Ramoops (aka. pstore) is general framework to dump logs into persistent RAM and which survive after a restart.
- Ramoops can dump:
  - Console message
  - Oops and panic log
  - Function tracing
- Ramoops function tracing is used to reverse program execution flow.
Demo for ramoops

**Step 1:** Enable all required kconfig options

CONFIG_PSTORE=y
CONFIG_PSTORE_FTRACE=y
CONFIG_DEBUG_FS=y

**Step 2:** Prepare for testing

Enable ramoops for function tracing

```
# mount -t debugfs debugfs /sys/kernel/debug/
# echo 1 > /sys/kernel/debug/pstore/record_ftrace
```

Enable watchdog

```
# echo 1 > /dev/watchdog
```

**Step 3:** Run testing case until system hangs

**Step 4:** Reboot with watchdog timeout

**Step 5:** Analyze tracing data

```
# mount -t pstore pstore /mnt
# cat /mnt/ftrace-ramoops-0 > tracing.log
```
Demo and ramoops log

CPU:0 ts:1000329 ffff0000082c87ec ffff0000082c9030 single_start <- seq_read+0x1a0/0x4c0
CPU:0 ts:1000330 ffff0000087214d4 ffff0000082c9058 dbg_ws_hang_show <- seq_read+0x1c8/0x4c0
CPU:0 ts:1000331 ffff0000080a428c ffff00000872151c __ioremap <- dbg_ws_hang_show+0x54/0xa8
CPU:0 ts:1000332 ffff0000080a41b0 ffff0000080a42a0 __ioremap_caller <- __ioremap+0x38/0x50
CPU:0 ts:1000333 ffff0000080a3c44 ffff0000080a41d8 pfn_valid <- __ioremap_caller+0x60/0xf0
CPU:0 ts:1000334 ffff0000082569c4 ffff0000080a3c4c memblock_is_map_memory <- pfn_valid+0x1c/0x30
CPU:0 ts:1000335 ffff0000082517f8 ffff0000080a41ec get_vm_area_caller <- __ioremap_caller+0x74/0xf0
CPU:0 ts:1000336 ffff000008251438 ffff00000825182c __get_vm_area_node <- get_vm_area_caller+0x54/0x68

https://youtu.be/2_SldeQrG-Y
Overview

● Discussion for practical scenarios
  ○ Finding program execution flow for system hang
  ○ CPU is dead, how to read current CPU state?
  ○ Offline analysis of program execution flow

● You can extend for your SoC
The scenario - CPU is dead, how to read current CPU state?

When a CPU dies, perhaps locked up with IRQ/FIQ masked or a stuck waiting for a bus access, the dead CPU cannot react to SMP inter-processor interrupt to dump backtrace.

JTAG debugger could be used to check dead CPU state, but we can explore a more convenient method in kernel for systems that have other CPUs are still alive so can dump PC value for dead CPU.

- **CORESIGHT_CPU_DEBUG**: Coresight CPU debug module allows Linux SMP partners to watch each other (enhanced LOCKUP_DETECTOR).
- Can be used to dump CPU state based on ARM sample-based profiling extension.
- Last CPU PC before failure is stored in its debug unit and other processors can extract this too (no cache problems)!
Demo for Coresight CPU debug

**Step 1:** Enable all required kconfig options

CONFIG_CORESIGHT_CPU_DEBUG=y

**Step 2:** Prepare for testing

Disable CPU idle states in command line (if need): nohlt

Enable panic on RCU stall if system doesn’t support hard lock detection:

```
# sysctl -w kernel.panic_on_rcu_stall=1
```

**Step 3:** Run test case until system panic

**Step 4:** Read debug info logged during panic
Demo video and Coresight CPU debug log

https://youtu.be/mFuvWrUrlwo
Overview

● Discussion for practical scenarios
  ○ Finding program execution flow for system hang
  ○ CPU is dead, how to know CPU current state?
  ○ Offline analysis of program execution flow

● You can extend for your SoC
The scenario - Offline analysis of program execution flow

If hang or panic issues happen in production release, we usually need to use analyse program execution flow offline using host tools.

Ramoops with function tracing is often not enabled for production testing because it might downgrade performance seriously; and thus alter or prevent reproduction for heisenbugs.

Coresight hardware tracer is a good candidate to record program execution flow with minimising overhead.

- Coresight + kdump can store hardware tracing data in dump file for kernel panic and we can rely on crash tool to extract tracing data
- Perf + OpenCSD tool decodes tracing data and outputs readable program flow
- It smoothly supports kernel panic debugging but so far it isn’t for debugging system hang
- Trick: If Coresight RAM is preserved after reset then useful Coresight tracing data can still be extracted for debugging hang issues
Demo for Coresight + Kdump

**Step 1:** Enable all required kconfig options

```bash
CONFIG_CORESIGHT=y
CONFIG_CORESIGHT_LINKS_AND_SINKS=y
CONFIG_CORESIGHT_LINK_AND_SINK_TMC=y
CONFIG_CORESIGHT_SOURCE_ETM4X=y
CONFIG_CORESIGHT_PANIC_KDUMP=y
CONFIG_KEXEC=y
CONFIG_CRASH_DUMP=y
```

**Step 2:** Prepare for testing

Enable Coresight tracer and sink
Use `kexec` to load capture-dump kernel and dtb

**Step 3:** Run test case until **system panic**

**Step 4:** Boot capture-dump kernel, save core file

**Step 5:** Extract Coresight tracing data with crash

```bash
# crash vmlinux vmcore
crash> extend csdump.so
crash> csdump out_folder
```

**Step 6:** Boot original kernel for analysis with `perf`

To avoid kernel build id mismatch when analysing coresight trace data, we can run original kernel with kernel symbol file:

```bash
# perf script -v -a -F cpu,event,ip,sym,symoff -i perf.data --kallsyms /proc/kallsyms
```
Demo video and perf decodes program flow

```bash
# ./perf script -f -v -a -F cpu,event,ip,sym,symoff --kallsyms /proc/kallsyms
build id event received for [kernel.kallsyms]: 32eeb5c9c99d00a63d0921cbd815c32385c36710
Using /proc/kcore for kernel object code
Using /proc/kallsyms for symbols
Frame deformatter: Found 4 FSYNCS
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
[000] branches: ffff000008972e7c arch_counter_get_cntpct+0xc
[000] branches: ffff000008afa6e8 __delay+0x90
[000] branches: ffff000008afa6dc __delay+0x84
https://youtu.be/oLIByzVGeFU
```
Overview

- Discussion for practical scenarios
  - Finding program execution flow for system hang
  - CPU is dead, how to know CPU current state?
  - Offline analysis of program execution flow

- You can extend for your SoC
You can extend for your SoC

You can extend to use other bus masters (DSP, MCU, etc) to access coresight CPU debug module to recover last PC in case all CPUs for Linux are dead.

You can extend other bus masters to support support a kdump-like workflow to preserve DDR and coresight trace data (preserved after reset) which can then be analysed using open source tools.
Related materials

- Coresight related patches and testing case
  - [https://git.linaro.org/people/leo.yan/linux-debug-workshop.git/log/?h=acme_perf_core_cs_dev](https://git.linaro.org/people/leo.yan/linux-debug-workshop.git/log/?h=acme_perf_core_cs_dev)
  - Ramoops and coresight CPU debug module has been merged into mainline kernel
  - Coresight for kdump supporting patches are working in progress

- This session adds new ideas not found in our previous debugging session:
  - **BUD17-TR04: Kernel Debug Stories**
  - Kernel Debug Stories covers far too many techniques to allow time for live demos ;-)}
Linaro Limited Lifetime Warranty

This training presentation comes with a lifetime warranty.

Everyone here today can send questions about today’s session and suggestions for future topics to support@linaro.org.

Member engineers can also use this address to get support on any other Linaro output. Engineers from club and core members are welcome to contact us to discuss how Support and Solutions Engineering can help you with additional services and training.

With thanks to Andrew Hennigan for introducing the idea of placing a guarantee on training.
Thank You

#HKG18
HKG18 keynotes and videos on: connect.linaro.org
For further information: support@linaro.org or www.linaro.org