Event Tracing and Pstore
with a pinch of Dynamic Debug

Sai Prakash Ranjan
saiprakash.ranjan@codeaurora.org
Overview

- Introduction to Register Trace Buffer (RTB)
- Limitations of RTB making it unsuitable for upstream
- Tracepoints and Event tracing
- IO tracepoint and read/write trace events as an alternative to RTB
- Extracting IO trace events of previous boot?
- Basics of Pstore and the IO trace events usecase
- Pstore logging of all trace events
- Filtering IO trace events based on subsystems
- Dynamic debug
- Future ideas
Introduction

- Started with Register Trace Buffer (RTB) developed by Laura Abbott for MSM kernels [1].
- Introduced nearly 8 years ago and still used today for debug in QCOM.
- Main use case is to debug abnormal resets due to unclocked access or watchdog bite where the data in the cache would get lost (may not be applicable for modern SoCs as flushing caches would be handled by the hardware itself).
- Powerful debug tool considering the use cases and the life span.

How it works:

- RTB stores events (mainly register reads and writes) in an uncached buffer and later these events are extracted from RAM dumps.
- Last few events logged per core before reset are very useful in the investigation for root cause.

Sample RTB logs extracted from Ramdump after crash is induced on SDM845 MTP [1]:

```
7502 [40.27s307]: 0x100203136: GICV_IRQ interrupt 13 handled from addr fffffff9d1e91ea0 gic_handle_irq
7502 [40.27s307]: 0x100203136: GICV_IRQ interrupt 14 handled from addr fffffff9d1e91ea0 gic_handle_irq
7502 [40.27s307]: 0x100203136: GICV_IRQ interrupt 15 handled from addr fffffff9d1e91ea0 gic_handle_irq
```

Limitations of RTB

● Design:
  ○ RTB as a separate kernel driver is redundant since Linux kernel already provides a good infrastructure called tracepoints for use cases like this where we need to statically add a hook in the function we would like to trace (that’s exactly what tracepoints are used for).
  ○ x86 has similar tracepoints for read/write_msr() calls as explained by Steven Rostedt [1]. So we can just use this existing infrastructure for our usecase of tracing register reads/writes in asm-generic code instead of introducing yet another interface for tracing.
  ○ Dependency on the external parser for trace decoding.

● Functional:
  ○ Not scalable. By default all register reads/writes are traced if RTB is enabled.
  ○ To filter the trace, duplicate `read{b,w,l,q}_no_log` apis used to filter the register reads/writes.
  ○ Filtering mainly based on log types rather than the log contents.

[1] https://lore.kernel.org/lkml/20180828120224.381d406a@gandalf.local.home/
Tracepoints

A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime. A tracepoint can be "on" (a probe is connected to it) or "off" (no probe is attached). When a tracepoint is "off" it has no effect, except for adding a tiny time penalty (checking a condition for a branch) and space penalty (adding a few bytes for the function call at the end of the instrumented function and adds a data structure in a separate section). When a tracepoint is "on", the function you provide is called each time the tracepoint is executed, in the execution context of the caller. When the function provided ends its execution, it returns to the caller (continuing from the tracepoint site).

Two elements are required for tracepoints:

- A tracepoint definition, placed in a header file.
- The tracepoint statement, in C code.

**Courtesy:** Linux Kernel Documentation: [Tracepoints](https://www.linux-kernel.org/doc/tracepoints.html)
IO Tracepoint and read/write trace events

/* Tracepoint for generic IO read/write, i.e., __raw__read/write([b,j,w,q]) */

DECLARE_EVENT_CLASS(io_trace_class, 
    TP_PROTO(const char *type, int cpu, u64 ts, void *addr, 
              unsigned long ret_ip), 
    TP_ARGS(type, cpu, ts, addr, ret_ip), 
    TP_STRUCT__entry( 
        __string( type, type ) 
        __field( int, ts ) 
        __field( void *, addr ) 
        __field( unsigned long, ret_ip ) 
    ), 
    TP_fast_assign( 
        __assign_str(type, type), 
        __entry->cpu = cpu; 
        __entry->ts = ts; 
        __entry->addr = addr; 
        __entry->ret_ip = ret_ip; 
    ), 
    TP_printk("%s:/cpu:%d ts:%d data:%d/k caller:%s", 
               __str(type), __entry->cpu, __entry->ts, __entry->addr, "\n") 
);

DEFINE_EVENT(io_trace_class, io_read, 
    TP_PROTO(const char *type, int cpu, u64 ts, void *addr, 
              unsigned long ret_ip), 
    TP_ARGS(type, cpu, ts, addr, ret_ip) 
);

DEFINE_EVENT(io_trace_class, io_write, 
    TP_PROTO(const char *type, int cpu, u64 ts, void *addr, 
              unsigned long ret_ip), 
    TP_ARGS(type, cpu, ts, addr, ret_ip) 
);
Enabling IO Tracepoint and read/write trace events

- Via kernel command line: trace_event=io for IO traces during boot.
- After boot:
  - `cd /sys/kernel/debug/tracing/events/io` (debugfs should be mounted)
  - `echo 1 > enable` (For read and write trace events)
  - `echo 1 > io_read-enable` or `echo 1 > io_write-enable` (For read/write trace events respectively)
Advantages of using IO trace events

- Trace events can be filtered in the kernel in number of ways which are described in linux kernel documentation [1], some of them below:
  - Filter expressions
  - Individual event filter
  - Subsystem filter (Kernel subsystem)
  - PID filtering
- Ftrace in general supports filtering based on function name, wildcards, commands and more.
- But we are more interested more in filtering events for driver subsystems (like gpu/usb/remoteproc) rather than kernel subsystems (scheduler, mm), more explained later.
- One of the advantages of using IO tracepoint instead of RTB driver is getting all the core ftrace features already present in ftrace and event tracing for free.

Extracting IO trace events of previous boot?

- Main usecase for having IO trace events is to debug abnormal resets.
- So unless there is a mechanism to get the logs of previous boot, only having the IO read/write events in the trace buffer is of less use.
- RTB downstream depends on the external parser and the ramdump for extracting these events but we need something in kernel to make developer’s life easier.
- What feature is already available in kernel which fits this usecase?
  - PSTORE
Pstore: Persistent storage for a kernel's "dying breath"

Pstore is a filesystem which provides a generic interface to capture kernel records in the dying moments or we could redefine it as a generic interface to capture kernel records that will persist across reboots.

Pstore supports different types of records. Some of the commonly used:

- DMESG: Dmesg logs shown only for oops/panic
- CONSOLE: Console logs
- FTRACE: Function trace logs, only function tracer supported currently
- PMSG: User space logs

More frontend records can be easily added to pstore.

For example: EVENT record addition to pstore for trace events

Persistent storage for a kernel's "dying breath" - https://lwn.net/Articles/434821/
Pstore Ramoops backend

- Ramoops is an oops/panic logger that writes its logs to RAM before the system crashes. Ramoops needs a system with persistent RAM so that the content of that area can survive after a restart.

- Better definition: Ramoops is a backend interface which enables pstore records to use persistent RAM as their storage to survive across reboots.

- Records are stored in following format in pstore filesystem and can be read after mounting pstore:
  - console-ramoops
  - dmesg-ramoops-N, where N is the record number
  - ftrace-ramoops
  - pmsg-ramoops-ID

- More info on using ramoops can be found here [1]

Pstore Ramoops

- Ramoops uses a predefined memory area to store the logs.
- Setting the ramoops parameters can be done through module parameters or through ramoops device tree node under reserved memory.
- We will look at DT node for ramoops for Dragonboard 410c board:

```c
reserved-memory {
    ramoops@bff00000{
        compatible = "ramoops";
        reg = <0x0 0xbff00000 0x0 0x100000>;
        record-size = <0x200000>;
        console-size = <0x200000>;
        ftrace-size = <0x200000>;
    }
};
```

- For events, the additional property would be **event-size**. Full bindings for ramoops here [1].

Why only IO trace events into pstore?

- We can easily extend support to log all trace events into pstore and not just IO trace events. Example below shows sched trace events of previous boot.

```bash
# trace_event=sched tp_pstore in command line
# reboot -f
# mount -t pstore pstore /sys/fs/pstore/
# tail /sys/fs/pstore/event-ramoops-0

sched_switch: prev_comm=swapperia 1 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm=rcu_preempt next_pid=10 next_prio=120
sched_switch: prev_comm=rcu_preempt prev_pid=10 prev_prio=120 prev_state=R+ ==> next_comm=swapperia 1 next_pid=0 next_prio=120
sched_waking: comm=rcu_sched pid=11 prio=120 target_cpu=002
sched_wakeup: comm=rcu_sched pid=11 prio=120 target_cpu=002
sched_switch: prev_comm=swapperia 2 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm=rcu_sched next_pid=11 next_prio=120
sched_switch: prev_comm=rcu_sched prev_pid=11 prev_prio=120 prev_state=R+ ==> next_comm=swapperia 2 next_pid=0 next_prio=120
sched_waking: comm=reboot pid=1867 prio=120 target_cpu=000
sched_wakeup: comm=reboot pid=1867 prio=120 target_cpu=000
sched_switch: prev_comm=swapperia 0 prev_pid=0 prev_prio=120 prev_state=S ==> next_comm=reboot next_pid=1867 next_prio=120
```
Filtering IO trace events based on subsystems

- Ftrace has a limitation where it has no file based filter mechanism in general and hence events cannot be filtered based on file path inputs unlike dynamic debug.
- Very high volume of traces generated from events like IO tracing need to have a filter based on files specified. In other words we need a filter based on driver subsystems, Ftrace already has filters for kernel subsystems like scheduler/mm etc.
- This can also be very helpful to narrow down issues to specific driver subsystems than having to look around everywhere.
- What can be used for this particular usecase?
  - DYNAMIC DEBUG
Dynamic debug

- Dynamic debug is designed to dynamically enable/disable kernel code to obtain additional kernel information. If CONFIG_DYNAMIC_DEBUG is set, then all pr_debug()/dev_dbg() and some other calls can be dynamically enabled per-callsite.

- There are several advantages of using dynamic debug:
  - Provides filtering based on below inputs using simple query language:
    - Source filename
    - Function name
    - Line number (including ranges of line numbers)
    - Module name
  - Provides a debugfs control file: <debugfs>/dynamic_debug/control for reading and controlling the behavior of debug statements.
  - Can be enabled during boot and module initialization time

- More usecases of dynamic debug is in kernel documentation [1]

Dynamic debug for IO trace events

For IO event tracing, we add a flag ‘e’ to filter events for specified input paths or driver subsystems.

- Ex: Enable IO event tracing in qcom soc dir during bootup: `dyndbg=file drivers/soc/qcom/* +e`
- Or after boot: `echo 'file drivers/soc/qcom/* +e' > sys/kernel/debug/dynamic_debug/control`

```
/# echo "file drivers/soc/qcom/* +e" > /sys/kernel/debug/dynamic_debug/control
/# cat sys/kernel/debug/dynamic_debug/control | grep -e
drivers/soc/qcom/qcom-ocp:182 [qcom_ocp] qmp_message_empty = "read"
drivers/soc/qcom/qcom-ocp:166 [qcom_ocp] qmp_close = "write"
drivers/soc/qcom/qcom-ocp:167 [qcom_ocp] qmp_close = "write"
drivers/soc/qcom/qcom-ocp:112 [qcom_ocp] qmp_send = "write"
drivers/soc/qcom/qcom-ocp:222 [qcom_ocp] qmp_send = "write"
drivers/soc/qcom/qcom-ocp:180 [qcom_ocp] qmp_link_acked = "read"
drivers/qcom/qcom-ocp:210 [qcom_ocp] qmp_ucore_channel_up = "read"
drivers/qcom/qcom-ocp:185 [qcom_ocp] qmp_mcore_channel_acked = "read"
drivers/qcom/qcom-ocp:75 [qcom_ocp] qmp_magic_valid = "read"
drivers/qcom/qcom-ocp:103 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:109 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:110 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:118 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:121 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:131 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:142 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:155 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-ocp:158 [qcom_ocp] qmp_open = "read"
drivers/qcom/qcom-geni-se.c:181 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:143 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:144 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
drivers/qcom/qcom-geni-se.c:142 [qcom_geni_sie] qmp_get_qmp_hw_version = "read"
```
/ #
Sample IO event tracing with pstore and dyndbg

1) Set command line with dyndbg, trace_event and tp_pstore parameter as below:

```bash
# dyndbg=file drivers/soc/qcom/* +e" trace_event=io tp_pstore
```

2) Bus hang by reading below debugfs entry with bus_hang module.

```bash
# cat /sys/kernel/debug/hang/bus_hang
```

3) After restart, we can find the cause in last entry i.e. (bus_hang_mdp+0xa4/0xb8)

```bash
# cat /sys/fs/pstore/event-ramoops-0
```

4) Offending register access found as below:

```bash
# (gdb)
# (gdb) list *(bus_hang_mdp+0xa4)
# 0xffff0000867cde8 is in bus_hang_mdp (drivers/soc/qcom/bus_hang.c:10).
# 5  static int bus_hang_mdp(void *data, u64 *val)
# 6  {
# 7   void *p = ioremap(0x01a01000, SZ_4K);
# 8   unsigned int a;
# 9   # 10   a = __raw_readl((void *)(unsigned long)p + 0x40)); ------
# 11   *val = a;
# 12   return 0;
# (gdb)
```
Future ideas

- Ftrace using Pstore storage: Idea by Joel Fernandes
  - Use pstore pages for ftrace data so that complete ftrace data is available on reset.

- Device Ramoops:
  - Capture device states or client driver data and save in pstore for post-mortem debugging.

- File or module based filtering support to Ftrace:
  - Can we use something similar to dynamic debug?
  - Just an idea, actual how to is still a ? - Ideas welcome
Any questions, feedbacks or suggestions?
Thank you

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

contactus@linaro.org