Event Tracing and Pstore

with a pinch of **Dynamic Debug**

Sai Prakash Ranjan
saiprakash.ranjan@codeaurora.org

Linaro connect
San Diego 2019

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.

[1] https://source.codeaurora.org/quic/la/kernel/msm-4.14/tree/kernel/trace/msm_rtb.c

RTB Example

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

```
7952 [48.273887] [8482003136] : LOGK_IRQ interrupt 13 handled from addr ffffff9d1be81aa0 gic_handle_irq
795a [48.273910] [8482003576] : LOGK_IRQ interrupt 4 handled from addr ffffff9d1be81b20 gic_handle_irq
7962 [48.273954] [8482004434] : LOGK_CTXID context id 1b called from addr ffffff9d1be86540 __switch_to
796a [48.273977] [8482004876] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
7972 [48.273996] [8482005226] : LOGK_CTXID context id 1d called from addr ffffff9d1be86540 __switch_to
797a [48.274023] [8482005743] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
7982 [49.015641] [8496244812] : LOGK_IRQ interrupt 0 handled from addr ffffff9d1be81b20 gic_handle_irq
798a [49.015655] [8496245079] : LOGK_CTXID context id 7 called from addr ffffff9d1be86540 __switch_to
7992 [49.015695] [8496245843] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
799a [49.020024] [8496328964] : LOGK_IRQ interrupt 13 handled from addr ffffff9d1be81aa0 gic_handle_irq
79a2 [49.020070] [8496329847] : LOGK_IRQ interrupt 4 handled from addr ffffff9d1be81b20 gic_handle_irq
79aa [49.020163] [8496331643] : LOGK_CTXID context id 159 called from addr ffffff9d1be86540 __switch_to
79b2 [49.020197] [8496332293] : LOGK_CTXID context id 52 called from addr ffffff9d1be86540 __switch_to
79ba [49.020235] [8496333028] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
79c2 [52.273887] [8558803143] : LOGK_IRQ interrupt 13 handled from addr ffffff9d1be81aa0 gic_handle_irq
79ca [52.273909] [8558803569] : LOGK_IRQ interrupt 4 handled from addr ffffff9d1be81b20 gic_handle_irq
79d2 [52.273953] [8558804415] : LOGK_CTXID context id 1b called from addr ffffff9d1be86540 __switch_to
79da [52.273977] [8558804860] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
79e2 [52.273995] [8558805206] : LOGK_CTXID context id 1d called from addr ffffff9d1be86540 __switch_to
79ea [52.274025] [8558805781] : LOGK_CTXID context id 52 called from addr ffffff9d1be86540 __switch_to
79f2 [52.274095] [8558807134] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
79fa [52.320121] [8559690828] : LOGK_IRQ interrupt 0 handled from addr ffffff9d1be81b20 gic_handle_irq
7a02 [52.320135] [8559691092] : LOGK_CTXID context id 56 called from addr ffffff9d1be86540 __switch_to
7a22 [52.320247] [8559693243] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
7a2a [56.273887] [8635603147] : LOGK_IRQ interrupt 13 handled from addr ffffff9d1be81aa0 gic_handle_irq
7a32 [56.273910] [8635603573] : LOGK_IRQ interrupt 4 handled from addr ffffff9d1be81b20 gic_handle_irq
7a3a [56.273954] [8635604419] : LOGK_CTXID context id 1b called from addr ffffff9d1be86540 __switch_to
7a42 [56.273977] [8635604861] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
7a4a [56.273995] [8635605211] : LOGK_CTXID context id 1d called from addr ffffff9d1be86540 __switch_to
7a52 [56.274025] [8635605794] : LOGK_CTXID context id 52 called from addr ffffff9d1be86540 __switch_to
7a5a [56.274096] [8635607149] : LOGK_CTXID context id 0 called from addr ffffff9d1be86540 __switch_to
7a62 [59.906110] [8705341817] : LOGK_IRQ interrupt 0 handled from addr ffffff9d1be81b20 gic_handle_irq
7a6a [59.906126] [8705342119] : LOGK_CTXID context id 24e called from addr ffffff9d1be86540 __switch_to
7a72 [59.908260] [8705383109] : LOGK_READL from address ffffff8010f3b040(5c00040) called from addr ffffff9d1c37bcb8 crash_mtp
```

```
#include <linux/debugfs.h>
#include <linux/module.h>
#include <asm/io.h>

static int crash_mtp(void *data, u64 *val)
{
        void *p = ioremap(0x5c00000, SZ_4K);
        unsigned int a;

        a = __raw_readl((void *)((unsigned long)p + 0x40));

        *val = a;

        return 0;
}

DEFINE_SIMPLE_ATTRIBUTE(crash_fops, crash_mtp, NULL, "%llu\n");

static int crash_init(void)
{
        struct dentry *dir;

        dir = debugfs_create_dir("crash", 0);
        if (!dir)
                return -1;

        debugfs_create_file("crash", 0644, dir, NULL, &crash_fops);

        return 0;
}
```

[1] https://github.com/saiprakash-ranjan/Bus-Hang/blob/master/crash_sdm845.c

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

IO Tracepoint and read/write trace events

```
/*
 * Tracepoint for generic IO read/write, i.e., __raw_{read,write}{b,l,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,                   cpu     )
                __field( u64,                   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("type=%s cpu=%d ts:%llu data=0x%lx caller=%pS",
                  __get_str(type), __entry->cpu, __entry->ts,
                  (unsigned long)__entry->addr, (void *)__entry->ret_ip)
);

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)
);
```

```
#define io_tracepoint_active(t) static_key_false(&(t).key)
extern void do_trace_io_write(const char *type, void *addr);
extern void do_trace_io_read(const char *type, void *addr);
#else
#define io_tracepoint_active(t) false
static inline void do_trace_io_write(const char *type, void *addr) {}
static inline void do_trace_io_read(const char *type, void *addr) {}
#endif /* CONFIG_TRACING_EVENTS_IO */

#define __raw_write(v, a, _l)     ({                                    \
        volatile void __iomem *_a = (a);                                \
        if (io_tracepoint_active(__tracepoint_io_write))                \
                do_trace_io_write(__stringify(write##_l), (void __force *)(_a));\
        arch_raw_write##_l((v), _a);                                    \
        })

#define __raw_writeb(v, a)      __raw_write((v), a, b)
#define __raw_writew(v, a)      __raw_write((v), a, w)
#define __raw_writel(v, a)      __raw_write((v), a, l)
#define __raw_writeq(v, a)      __raw_write((v), a, q)

#define __raw_read(a, _l, _t)   ({                                      \
        _t __a;                                                         \
        const volatile void __iomem *_a = (a);                          \
        if (io_tracepoint_active(__tracepoint_io_read))                 \
                do_trace_io_read(__stringify(read##_l), (void __force *)(_a));  \
        __a = arch_raw_read##_l(_a);                                    \
        __a;                                                            \
        })

#define __raw_readb(a)__raw_read((a), b, u8)
#define __raw_readw(a)          __raw_read((a), w, u16)
#define __raw_readl(a) __raw_read((a), l, u32)
#define __raw_readq(a)__raw_read((a), q, u64)

void do_trace_io_write(const char *type, void *addr)
{
        trace_io_write(type, raw_smp_processor_id(), sched_clock(), addr,
                       _RET_IP_);
}

void do_trace_io_read(const char *type, void *addr)
{
        trace_io_read(type, raw_smp_processor_id(), sched_clock(), 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)

```
/sys/kernel/debug/tracing # cat trace
# tracer: nop
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [000] d.h2  1299.071007: io_read: type=readl cpu=0 ts:1299071006776 data=0xffffff8008970800 caller=qcom_geni_serial_isr+0x504/0x5d4
          <idle>-0     [000] d.h2  1299.071012: io_read: type=readl cpu=0 ts:1299071011411 data=0xffffff8008970614 caller=qcom_geni_serial_stop_tx+0x194/0x224
          <idle>-0     [000] d.h2  1299.071015: io_write: type=writel cpu=0 ts:1299071014432 data=0xffffff800897080c caller=qcom_geni_serial_stop_tx+0x1a8/0x224
          <idle>-0     [000] d.h2  1299.071015: io_write: type=writel cpu=0 ts:1299071015214 data=0xffffff8008970614 caller=qcom_geni_serial_stop_tx+0x1bc/0x224
          <idle>-0     [000] d.h2  1299.071017: io_read: type=readl cpu=0 ts:1299071017193 data=0xffffff8008970040 caller=qcom_geni_serial_stop_tx+0x1d0/0x224
          <idle>-0     [000] d.h2  1299.071026: io_read: type=readl cpu=0 ts:1299071021724 data=0xffffff8008970614 caller=qcom_geni_serial_isr+0x58c/0x5d4
       kworker/6:0-44   [006] d..2  1299.276601: io_write: type=writel cpu=6 ts:1299276599381 data=0xffffff8009d05780 caller=pmic_arb_read_cmd+0x1c4/0x1cc
       kworker/6:0-44   [006] d..2  1299.276605: io_read: type=readl cpu=6 ts:1299276604329 data=0xffffff8009d05788 caller=pmic_arb_wait_for_done+0x13c/0x140
       kworker/6:0-44   [006] d..2  1299.276608: io_read: type=readl cpu=6 ts:1299276608391 data=0xffffff8009d05788 caller=pmic_arb_wait_for_done+0x13c/0x140
       kworker/6:0-44   [006] d..2  1299.276613: io_read: type=readl cpu=6 ts:1299276612506 data=0xffffff8009d05788 caller=pmic_arb_wait_for_done+0x13c/0x140
       kworker/6:0-44   [006] d..2  1299.276615: io_read: type=readl cpu=6 ts:1299276615318 data=0xffffff8009d05798 caller=pmic_arb_read_data.isra.1+0x9c/0xb0
       kworker/6:0-44   [006] d..2  1299.276618: io_read: type=readl cpu=6 ts:1299276618027 data=0xffffff8009d0579c caller=pmic_arb_read_data.isra.1+0x9c/0xb0
       kworker/6:0-44   [006] d..2  1299.276621: io_write: type=writel cpu=6 ts:1299276621360 data=0xffffff800caf0010 caller=pmic_arb_write_data.isra.0+0xa4/0xbc
       kworker/6:0-44   [006] d..2  1299.276623: io_write: type=writel cpu=6 ts:1299276623391 data=0xffffff800caf0014 caller=pmic_arb_write_data.isra.0+0xa4/0xbc
       kworker/6:0-44   [006] d..2  1299.276626: io_write: type=writel cpu=6 ts:1299276625422 data=0xffffff800caf0000 caller=pmic_arb_write_cmd+0x1c4/0x1c8
       kworker/6:0-44   [006] d..2  1299.276628: io_read: type=readl cpu=6 ts:1299276627402 data=0xffffff800caf0008 caller=pmic_arb_wait_for_done+0x13c/0x140
```

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.

[1] https://www.kernel.org/doc/html/latest/trace/events.html

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]

[1] https://www.kernel.org/doc/html/latest/admin-guide/ramoops.html

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:

```
reserved-memory {
        ramoops@bff00000{
                compatible = "ramoops";
                reg = <0x0 0xbff00000 0x0 0x100000>;

                record-size = <0x20000>;
                console-size = <0x20000>;
                ftrace-size = <0x20000>;
        };
};
```

Required properties:

- compatible: must be "ramoops"

- reg: region of memory that is preserved between reboots

Optional properties:

- ecc-size: enables ECC support and specifies ECC buffer size in bytes (defaults to 0: no ECC)

- record-size: maximum size in bytes of each dump done on oops/panic (defaults to 0: disabled)

- console-size: size in bytes of log buffer reserved for kernel messages (defaults to 0: disabled)

- ftrace-size: size in bytes of log buffer reserved for function tracing and profiling (defaults to 0: disabled)

- pmsg-size: size in bytes of log buffer reserved for userspace messages (defaults to 0: disabled)

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

[1] https://www.kernel.org/doc/Documentation/devicetree/bindings/reserved-memory/ramoops.txt

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.

```
# 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=swapper/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=swapper/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=swapper/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=swapper/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=swapper/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]

[1] https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html

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/aoss-qmp.c:182 [aoss_qmp]qmp_message_empty =e "readl"
drivers/soc/qcom/aoss-qmp.c:166 [aoss_qmp]qmp_close =e "writel"
drivers/soc/qcom/aoss-qmp.c:167 [aoss_qmp]qmp_close =e "writel"
drivers/soc/qcom/aoss-qmp.c:212 [aoss_qmp]qmp_send =e "writel"
drivers/soc/qcom/aoss-qmp.c:222 [aoss_qmp]qmp_send =e "writel"
drivers/soc/qcom/aoss-qmp.c:80 [aoss_qmp]qmp_link_acked =e "readl"
drivers/soc/qcom/aoss-qmp.c:90 [aoss_qmp]qmp_ucore_channel_up =e "readl"
drivers/soc/qcom/aoss-qmp.c:85 [aoss_qmp]qmp_mcore_channel_acked =e "readl"
drivers/soc/qcom/aoss-qmp.c:75 [aoss_qmp]qmp_magic_valid =e "readl"
drivers/soc/qcom/aoss-qmp.c:103 [aoss_qmp]qmp_open =e "readl"
drivers/soc/qcom/aoss-qmp.c:109 [aoss_qmp]qmp_open =e "readl"
drivers/soc/qcom/aoss-qmp.c:110 [aoss_qmp]qmp_open =e "readl"
drivers/soc/qcom/aoss-qmp.c:117 [aoss_qmp]qmp_open =e "readl"
drivers/soc/qcom/aoss-qmp.c:118 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/aoss-qmp.c:121 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/aoss-qmp.c:131 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/aoss-qmp.c:142 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/aoss-qmp.c:155 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/aoss-qmp.c:158 [aoss_qmp]qmp_open =e "writel"
drivers/soc/qcom/qcom-geni-se.c:181 [qcom_geni_se]geni_se_get_qup_hw_version =e "readl"
drivers/soc/qcom/qcom-geni-se.c:413 [qcom_geni_se]geni_se_config_packing =e "writel"
drivers/soc/qcom/qcom-geni-se.c:414 [qcom_geni_se]geni_se_config_packing =e "writel"
drivers/soc/qcom/qcom-geni-se.c:417 [qcom_geni_se]geni_se_config_packing =e "writel"
drivers/soc/qcom/qcom-geni-se.c:418 [qcom_geni_se]geni_se_config_packing =e "writel"
drivers/soc/qcom/qcom-geni-se.c:429 [qcom_geni_se]geni_se_config_packing =e "writel"
drivers/soc/qcom/qcom-geni-se.c:633 [qcom_geni_se]geni_se_tx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:634 [qcom_geni_se]geni_se_tx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:635 [qcom_geni_se]geni_se_tx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:636 [qcom_geni_se]geni_se_tx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:637 [qcom_geni_se]geni_se_tx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:666 [qcom_geni_se]geni_se_rx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:667 [qcom_geni_se]geni_se_rx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:668 [qcom_geni_se]geni_se_rx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:670 [qcom_geni_se]geni_se_rx_dma_prep =e "writel"
drivers/soc/qcom/qcom-geni-se.c:671 [qcom_geni_se]geni_se_rx_dma_prep =e "writel"
/ #
```

Sample IO event tracing with pstore and dyndbg

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

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

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

```
# 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)

```
# cat /sys/fs/pstore/event-ramoops-0
  io_write: type=writel cpu=0 ts:1423426774 data=0xffff00000d5065a4 caller=qcom_smsm_probe+0x52c/0x678
  io_write: type=writel cpu=0 ts:1423649847 data=0xffff00000d506608 caller=qcom_smsm_probe+0x52c/0x678
  io_read: type=readl cpu=1 ts:53095994171 data=0xffff00000a51d040 caller=bus_hang_mdp+0xa4/0xb8
```

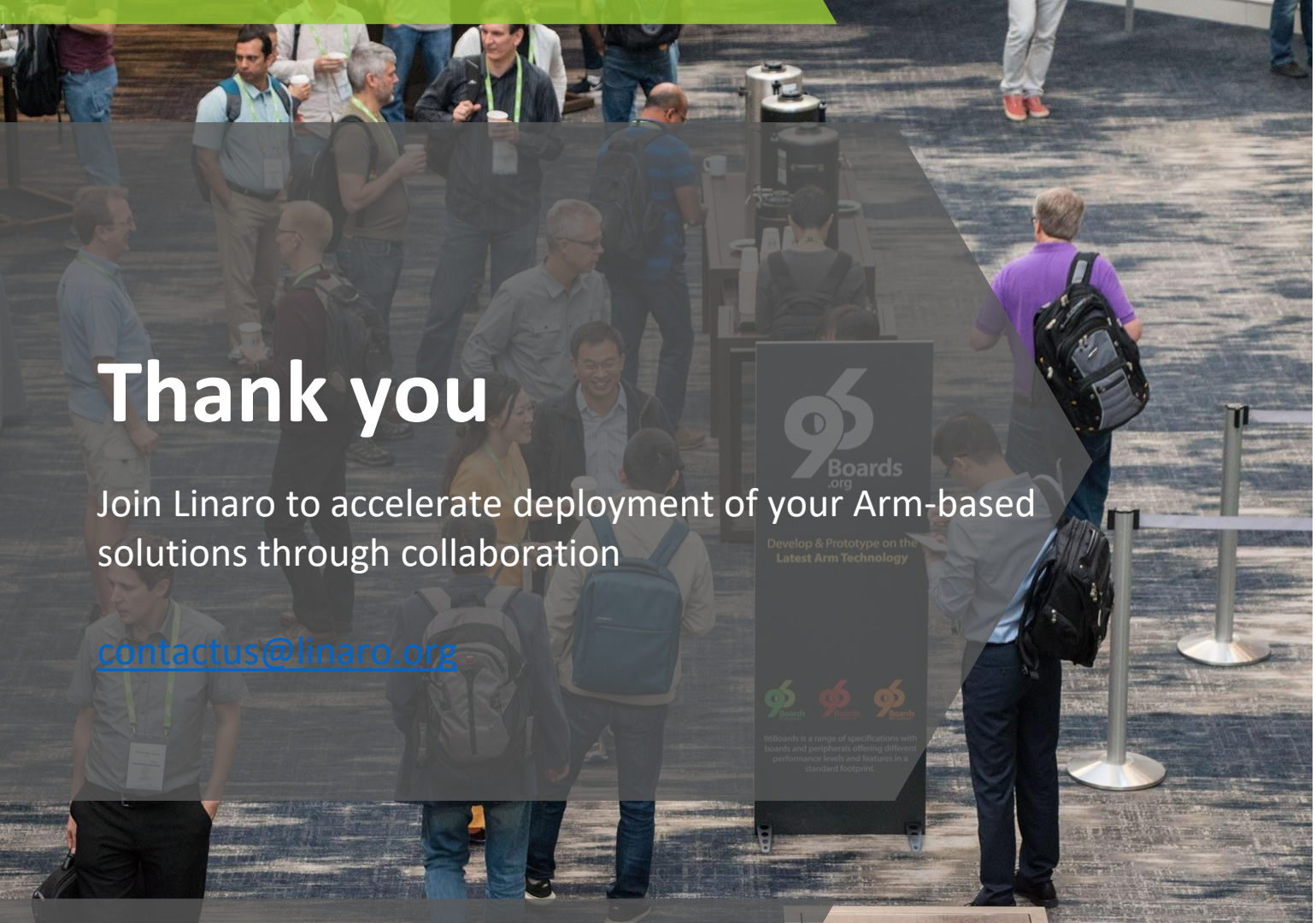
4) Offending register access found as below:

```
# (gdb)
  # (gdb) list *(bus_hang_mdp+0xa4)
  # 0xffff00000867cdc8 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
  # 12           *val = a;
  # 13
  # 14           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

Linaro connect
San Diego 2019