Diving into Linux Perf Ring Buffer

Leo Yan <*leo.yan@linaro.org*>
May 25, 2021

Abstract

Linux perf ring buffer is not only used to transfer the PMU event data, it's also a fundamental mechanism for hardware tracing with Intel PT, Arm CoreSight, etc. Therefore, the ring buffer implementation is critical and very challenge, it is required to provide high throughput, and should avoid causing any significant overload by itself for the buffer's management.

The purpose of this article is to provide a material if anyone wants to understand the internal of pref ring buffer. By diving into the details, it explains the perf ring buffer implementation and how the ring buffer is applicated in practice.

Keywords: Linux, perf, ring buffer, throughput, profiling

Introduction

Perf tool is a main stream profiling tool which is widely used in Linux community. At the early time, it was originally designed to support CPU PMU events, like CPU cycles, cache access and misses events, etc; afterwards, it was extended to support timers, software events (E.g. Ftrace tracepoints). Nowdays, it has integrated with the hardware trace and even can co-work with eBPF tracing.

To support these kind events, especially if developers want to record multiple events in one go, the ring buffer plays a critical role for event recording, the kernel and perf tool in the user space use the ring buffer to exchange data, and at the end stores record into data file.

The throughput is a big challenge in the implementation, particularly, from the performance pespective, it would be very interesting to know how the buffer is synchronized between user space and kernel, and how to support SMP if the buffer is shared by multiple CPUs. This article tries to dive into these details and give out answers to cate our curiosity.

In the sequential sections, the content is arranged as:

- The introduction for basic algorithm of the ring buffer;

- The mechanism for AUX ring buffer;
- At last, using Arm CoreSight as an example to explain how the ring buffer works with hardware trace.

Implementation for the ring buffer

Basic algorithm

As said in the textbook, the ring buffer should be managed by a head pointer and a tail pointer; the head pointer is manipulated by a writer and the tail pointer is updated by a reader respectively.

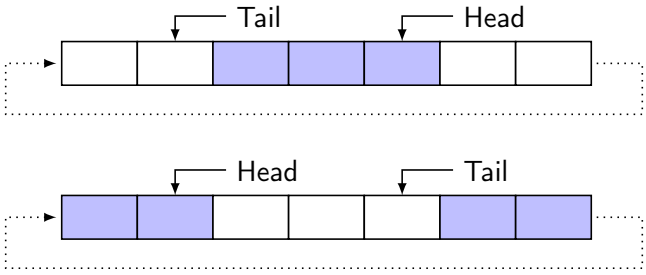

Figure 1: Ring buffer (without and with overflow)

Similarily, perf uses the same way to manage the ring buffer. There have two actors: a page contains the control structure for ring buffer management, and the ring buffer; following the naming convention, the page containing the control structure is named as "user page". The user page and the ring buffer are mapped to user space via the virtual mapping area (VMA) in the continuous address space, and the user page is mapped prior to the buffer.

The control structure is defined as `perf_event_mmap_page` which contains the head pointer `data_head` and the tail pointer `data_tail`. When the kernel starting to fill records into the ring buffer, it modifies the head pointer to reserve the memory so that later can safely store events into the buffer; on the other side, the perf tool updates the tail pointer when consumes data from the ring buffer.

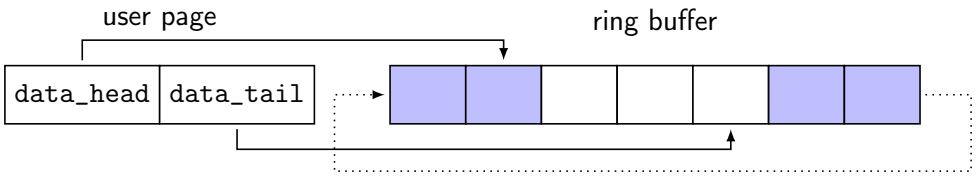

Figure 2: Perf ring buffer

Though the kernel allocates at once for all memory pages, including a dedicated page for the user page and the sequential pages for ring buffer, it's deferred to map the the pages to VMA area until the perf tool accesses the buffer from the user space. In other words, the kernel event subsystem uses the linear kernel virtual address for accessing the ring buffer, the perf tool in the user space accesses the ring buffer via VMA, and at the first time accesses the buffer's page, a data abort exeception for page fault is taken and the kernel

uses this occasion to map the page into VMA, thus the perf tool can continue to access the page after return back from page fault.

The function `perf_mmap_fault()` is invoked for handling page fault, which uses the function `perf_mmap_to_page()` to figure out which page should be mapped. If `pg_off` is zero it returns the pointer for the ring buffer's user page, otherwise it finds out the ring buffer's data page by using `pg_off-1` as the page index (since the first page in VMA is reserved for user page, it produces offset '1' between the VMA index and data pages index).

```
static vm_fault_t perf_mmap_fault(struct vm_fault *vmf)
{
        struct perf_event *event = vmf->vma->vm_file->private_data;
        struct perf_buffer *rb;
        vm_fault_t ret = VM_FAULT_SIGBUS;

        if (vmf->flags & FAULT_FLAG_MKWRITE) {
                if (vmf->pgoff == 0)
                        ret = 0;
                return ret;
        }

        rcu_read_lock();
        rb = rcu_dereference(event->rb);
        if (!rb)
                goto unlock;

        if (vmf->pgoff && (vmf->flags & FAULT_FLAG_WRITE))
                goto unlock;

        vmf->page = perf_mmap_to_page(rb, vmf->pgoff);
        if (!vmf->page)
                goto unlock;

        get_page(vmf->page);
        vmf->page->mapping = vmf->vma->vm_file->f_mapping;
        vmf->page->index = vmf->pgoff;

        ret = 0;
unlock:
        rcu_read_unlock();

        return ret;
}
```

Ring buffer allocation for different modes

Perf profiles programs with different modes: per thread mode, per cpu mode, and system wide mode; the question is how the ring buffer is organized for these modes. This section describes what's exactly these modes and how the ring buffer can meet the requirement for them; at last we can review if there have any race conditions for the ring buffer caused in these modes.

Per thread mode

For the per-thread mode, by specifying option `--per-thread` in perf command, the ring buffer is allocated for every profiled thread. In this mode when the profiled thread is scheduled on a CPU, the events on that CPU will be enabled; and if the thread is cheduled out from the CPU, the events on the CPU will be disabled. And if the thread is migrated from one CPU to another CPU, the events will be disabled on the previous CPU and enabled on the next CPU correspondingly.

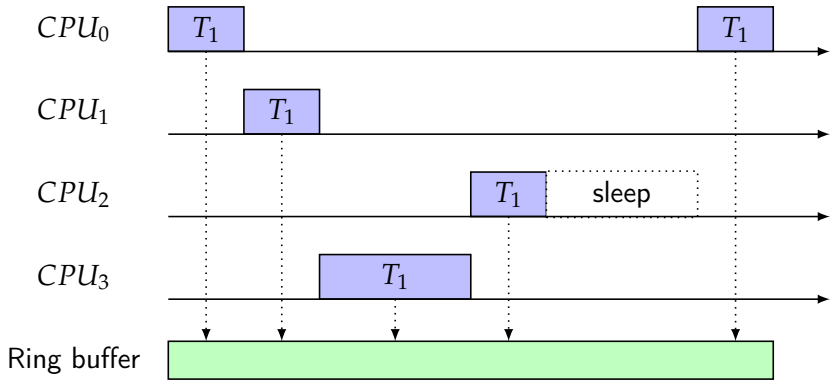

Figure 3: Ring buffer for per-thread mode

As shown in the figure 3, when perf runs in per-thread mode, a ring buffer is allocated for the profiled thread T_1. The ring buffer is dedicated for the thread T_1, and no matter the thread is scheduled on which CPUs. If the thread T_1 is running, the perf events will be recorded into the ring buffer; during the thread's sleeping period, all associated events will be disabled, thus the ring buffer will not record any trace data.

Per CPU mode

The option `-C` is used to collect samples only on the list of CPUs, the ring buffers are allocated for the specified CPUs. For the example in figure 4, the perf command receives option `-C 0,2`, as the result, two ring buffers serve for CPU0 and CPU2 separately.

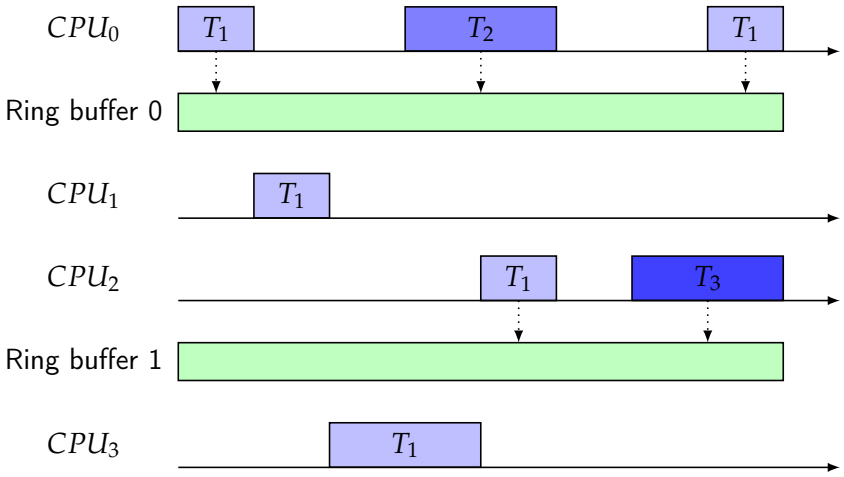

Figure 4: Ring buffer for per-cpu mode

On the other hand, in this example even there have tasks running on CPU1 and CPU3, since the ring buffer is absent for these two CPUs, any activities on them will be ignored.

A usage case is to combine the options for per-thread mode and per-CPU mode, e.g. the option `-C 0,2 --per-thread` is specified together; in this case, samples are recorded only when the profiled thread is scheduled on any of the listed CPUs.

System wide mode

By default if without specifying mode, or explicitly using option `-a` or `--all-cpus`, perf collects samples on all CPUs, with the system wide mode.

In this mode, every CPU has its dedicated ring buffer; the interested events will be always enabled and sampled on every CPU, thus all threads are monitored, and the samples are recorded into the ring buffer belonging to the CPU which the events occurred on.

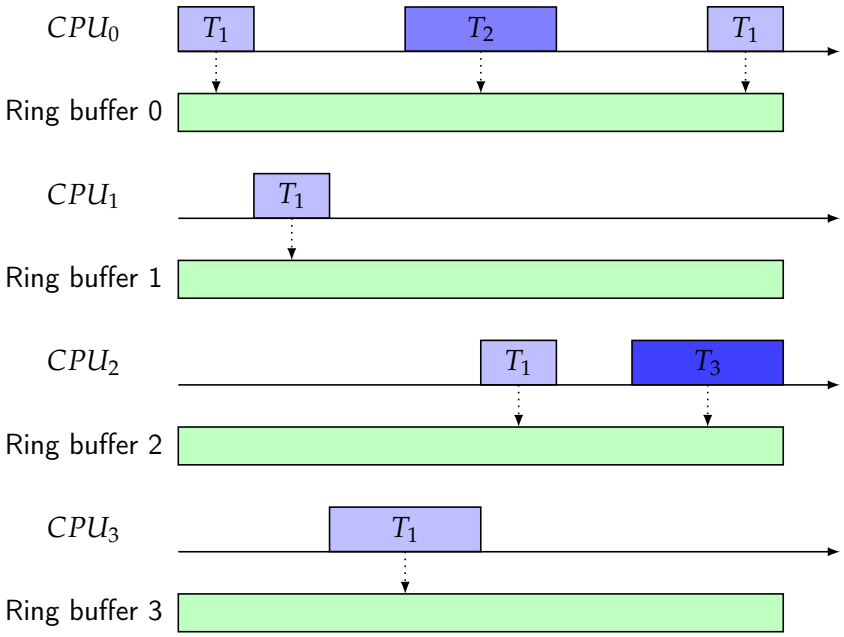

Figure 5: Ring buffer for system wide mode

Writing and reading buffer

From the previous section we get to know how the ring buffer is allocated for different modes, based on that this section will explain how the ring buffer is written or read.

Producer-consumer model

It's a typical producer-consumer model for using the ring buffer. In the Linux kernel, the events can produce samples which are stored into the ring buffer; the perf in user space consumes the samples by reading out the data from the ring buffer.

There have many occasions for generating samples: PMU interrupt handler when detecting event overflow, scheduling, dynamic tracepoint with

kprobe/uprobe, static tracepoint, etc. When a sample is recorded into the ring buffer, the kernel event core layer will wake up the thread which is polling on the events; perf works as a process to poll on the event, it will be waken up to read samples from the ring buffer.

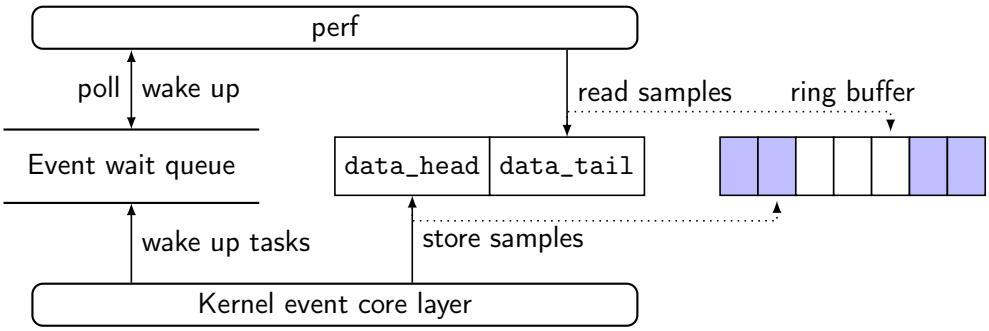

Figure 6: Writing and reading the ring buffer

Note, perf sleeps on the wait queue of the events rather than wait on the ring buffer; the ring buffer itself doesn't provide wait queue. But the event core layer in kernel is not only to wake up the sleeping tasks based on the event, on the other hand, it wakes up tasks based on the whole ring buffer.

Because multiple events share the same ring buffer for recording samples, when any event sample is stored into the ring buffer, the kernel event core layer simply wakes up sleeping tasks relevant to the ring buffer. This is fulfilled by the kernel function `ring_buffer_wakeup()` which iterates every event associated to the ring buffer and wakes up tasks on the wait queue of the events.

```
static void ring_buffer_wakeup(struct perf_event *event)
{
   struct perf_buffer *rb;

   rcu_read_lock();
   rb = rcu_dereference(event->rb);
   if (rb) {
      list_for_each_entry_rcu(event, &rb->event_list, rb_entry)
         wake_up_all(&event->waitq);
   }
   rcu_read_unlock();
}
```

Perf works as a standalone process; and after its process is waken up, it starts to check the ring buffers one by one, if finds any ring buffer contains samples it will read out the samples for statistics.

We can get to know that the perf process is possible to run on any CPUs, this leads to the ring buffer can be accessed from multiple CPUs simultaneously, which causes race conditions and should be handled properly. The details for handling race condition will be depicted later.

Writing samples into buffer

When an event couter is overflow, a sample will be taken and saved into the ring buffer; the function `__perf_event_output()` is used to fill samples into the ring buffer.

```
static __always_inline int
__perf_event_output(struct perf_event *event,
                    struct perf_sample_data *data,
                    struct pt_regs *regs,
                    int (*output_begin)(struct perf_output_handle *,
                                        struct perf_sample_data *,
                                        struct perf_event *,
                                        unsigned int))
{
        struct perf_output_handle handle;
        struct perf_event_header header;
        int err;

        /* protect the callchain buffers */
        rcu_read_lock();

        perf_prepare_sample(&header, data, event, regs);

        err = output_begin(&handle, data, event, header.size);
        if (err)
                goto exit;

        perf_output_sample(&handle, &header, data, event);

        perf_output_end(&handle);

exit:
        rcu_read_unlock();
        return err;
}
```

These main functions are called for outputting a sample:

- As its name indicates, the function `perf_prepare_sample()` prepares sample fields based on the sample type;
- `output_begin()` is a function pointer, it's passed dynamically via the argument for different writing direction, its purpose is to prepare the info for writing ring buffer, when return back the ring buffer info is stored in structure `perf_output_handle`;
- `perf_output_sample()` outputs the sample fields into the ring buffer;
- `perf_output_end()` updates the head pointer for user page so perf tool can see the latest value.

Let's look into `output_begin()`. Since the ring buffer supports two directions for writing: backward or forward, thus this function pointer is assigned according to the buffer's writing type, it can be `perf_output_begin_forward()` or `..._backward()` variant.

For the backward ring buffer due to the user page is mapped without 'PROT_WRITE', the tool in user space has no chance to update tail pointer, therefore, this case uses only head pointer and doesn't use the tail pointer. For the backward ring buffer, the head pointer points to the start of a sample, perf tool can read out the samples one by one based on sample's event size.

Alternatively, the forward ring buffer uses both head pointer and tail pointer for the buffer management, which is more often used in perf tool and matches the description in the section "Basic algorithm". To simplify the description, below uses the forward type of ring buffer to explain how to write samples.

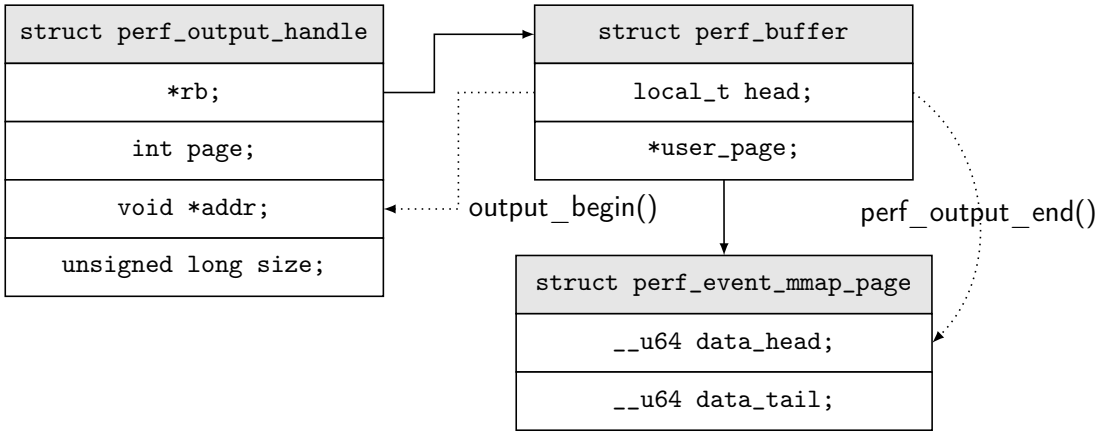

Figure 7: Structures for writing ring buffer

The event core layer in kernel uses the structure `perf_buffer` to track the buffer's latest header, and it keeps the information for buffer pages. If considering the life cycle, the structure `perf_buffer` is allocated once the ring buffer is created, and it's released when the ring buffer is destroyed.

It's possible that multiple events to write buffer concurrently. For instance, one software event and one hardware PMU event both are enabled for profiling, when the software event is in the middle of sampling, the hardware event maybe overflow and its interrupt is triggered in this case. This leads to the race condition for `perf_buffer::head`, because the software event sampling can be interrupted by the hardware event sampling.

Linux kernel uses compare-and-swap atomicity `local_cmpxchg()` to implement the locklesss mechanism for protecting `perf_buffer::head` (copied from `__perf_output_begin()`):

```
do {
        tail = READ_ONCE(rb->user_page->data_tail);
        offset = head = local_read(&rb->head);
        if (!rb->overwrite) {
                if (unlikely(!ring_buffer_has_space(head, tail,
                                                    perf_data_size(rb),
                                                    size, backward)))
                        goto fail;
        }

        /*
```

```
         * The above forms a control dependency barrier separating the
         * @tail load above from the data stores below. Since the @tail
         * load is required to compute the branch to fail below.
         *
         * A, matches D; the full memory barrier userspace SHOULD issue
         * after reading the data and before storing the new tail
         * position.
         *
         * See perf_output_put_handle().
         */

        if (!backward)
                head += size;
        else
                head -= size;
} while (local_cmpxchg(&rb->head, offset, head) != offset);
```

Comparing to `perf_buffer`, the structure `perf_output_handle` is only used during the event sampling. When an event is overflow, it firstly uses the structure `perf_output_handle` to establish a context for buffer's accessing, afterwards, sampling only uses the field `perf_output_handle::addr` as the destination address for copying sample data.

A significant benefit from the structure `perf_output_handle` is it allows different events to write buffer concurrently. Following the previous example, two copies of `perf_output_handle` work as two contexts for the software event and hardware event separately, thus every event reserves its own memory space in the function `out_begin()` and `perf_output_handle::addr` is used for filling samples.

Until the sample data has been stored into buffer, the ring buffer's header is synced from `perf_buffer::head` to `perf_event_mmap_page::data_head` in the function `perf_output_end()`. This delivers the message to the perf tool that "now you are safe to read out the new samples from the user space".

Reading samples from buffer

Like kernel's `perf_output_handle`, the structure `perf_mmap` maintains a context for ring buffer in the user space, the context contains the info for buffer's start address, end address and mask; these values can be used to calculate the buffer pointer and size to be read based on different buffer modes.

The pointers and buffer are different data objects, it's possible for out-of-order accessing to them on the modern CPUs with relaxed memory model. Furthermore, given perf and the profiled program running in multi-threads on multiple CPUs, the sequence between accessing pointers and buffer data must be promised by memory synchronization.

Besides the kernel function `perf_output_put_handle()`, two helpers `ring_buffer_read_head()` and `ring_buffer_write_tail()` are introduced with memory barriers in the user space, they cooperates to ensure the data dependency; the rationale for the memory synchronization is:

```
Kernel

if (LOAD ->data_tail) {
                    (A)
   STORE $data
   smp_wmb()       (B)
   STORE ->data_head
}
```

```
User space

LOAD ->data_head
smp_rmb()       (C)
LOAD $data
smp_mb()        (D)
STORE ->data_tail
```

The comment in source file `include/linux/ring_buffer.h` gives very nice description, here just tried to give explaination in my own words.

`(A)` indicates a control dependency between checking pointer `data_tail` and filling sample, so that the kernel has chance to decide if the buffer has enough free space or not. `(D)` needs to separate the ring buffer data reading from writing the pointer `data_tail`, perf tool firstly reads samples and then tells kernel that samples have been consumed and it can safely use it from then on. Since one of these two operations is reading and another is writing, thus it needs a full memory barrier. `(B)` is a writing barrier in the middle of two writing operations, which makes sure the head pointer updating must be later than finishing current sampling. `(C)` is a read memory barrier to split two read operations, it promises the program sequence for fetching the latest head pointer prior to reading samples.

```
static inline u64 ring_buffer_read_head(struct perf_event_mmap_page *base)
{
/*
 * Architectures where smp_load_acquire() does not fallback to
 * READ_ONCE() + smp_mb() pair.
 */
#if defined(__x86_64__) || defined(__aarch64__) || defined(__powerpc64__) || \
    defined(__ia64__) || defined(__sparc__) && defined(__arch64__)
       return smp_load_acquire(&base->data_head);
#else
       u64 head = READ_ONCE(base->data_head);

       smp_rmb();
       return head;
#endif
}

static inline void ring_buffer_write_tail(struct perf_event_mmap_page *base,
                                          u64 tail)
{
       smp_store_release(&base->data_tail, tail);
}
```

Some architectures support one-way permeable barrier with load-acquire and store-release operations, these barriers are more relax to impose the barrier on any irrelevant memory operations, so `(C)` and `(D)` can be optimized to use barriers `smp_load_acquire()` and `smp_store_release()` respectively.

If an architecture doesn't support load-acquire and store-release, it will roll-back to the old fashion with common memory barriers. A minor improvement

should be mentioned, as shown in `ring_buffer_read_head()`, if an architecture doesn't support load-acquire primitive, it doesn't simply fallback to `READ_ONCE() + smp_mb()` encapsulated in `smp_load_acquire()`, the reason is it wants to benefit the performance by using a more lightweight barrier `smp_rmb()` rather than `smp_mb()`.

The mechanism for AUX ring buffer

This chapter will examine the implementation for AUX ring buffer. It firstly concludes the relationship between the AUX ring buffer and the generic ring buffer; then it reviews how the AUX ring buffer co-work with the generic ring buffer, and what's the extension introduced by the AUX ring buffer for the sampling mechanism.

The relationship with the generic ring buffer

In an overview, the AUX ring buffer is an auxiliary for the generic ring buffer. The generic ring buffer is primarily used to store the event samples, and every event fromat complies with the definition in the union `perf_event`; the AUX ring buffer is for recording the hardware tracing, and the trace data format is hardware IP dependent. The advantage of introducing AUX ring buffer is it can de-couple the data transferring between the generic perf events and the hardware tracing.

It's nature for the AUX ring buffer to reuse the same algorithm with the generic ring buffer for the buffer management. The control stucture `perf_event_mmap_page` extends the new fields `aux_head` and `aux_tail` for the head and tail pointers of the AUX ring buffer.

The record option `record_opts::auxtrace_mmap_pages` is set during the AUX trace initialisation, otherwise, if perf session has not attached to any AUX trace, this option is the default value '0'. When it is a non-zero value, the function `auxtrace_mmap__mmap()` invokes system call mmap() and the kernel function `rb_alloc_aux()` serves for allocating pages; these pages will be deferred to map into VMA when detects the page fault which is the exactly same mechanism with the generic ring buffer. As the result, the perf tool have two types of ring buffer and needs to manage every ring buffer individually.

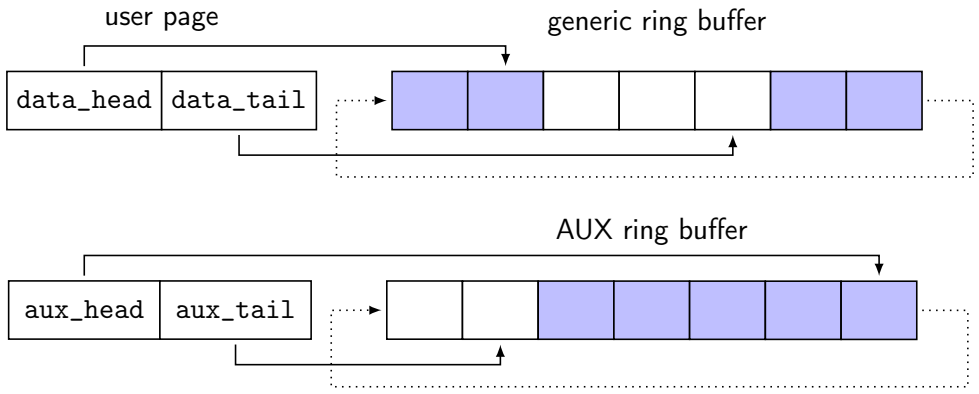

Figure 8: Perf generic ring buffer and AUX ring buffer

An interesting question is: Is it possible to only allocate the AUX ring buffer for hardware trace if there have no any other PMU events opened? Say below command only enables Arm CoreSight event:

```
perf record -e cs_etm/@tmc_etr0/u -a program
```

In fact, the generic ring buffer and AUX ring buffer are allocated in pairs, even when only the hardware trace event is enabled (see `mmap__mmap()`). The reason is the generic ring buffer is not merely for event sampling, it also records the process memory mapping so that the info can be used for parsing symbols, and the AUX event will be saved into the generic ring buffer as well. The AUX event and AUX trace data will be addressed in details in the later section.

Now let's review the AUX ring buffer deployment for perf modes. I.e., for per thread mode, perf tool only allocates one generic ring buffer and one AUX ring buffer for the whole session; for the system wide mode, perf allocates the generic ring buffer and the AUX ring buffer per CPU wise. The figure demonstrates the buffers layout in the system wide mode; if there have any activities on one CPU, the event samples and the hardware trace data are recorded into the dedicated buffers for the CPU.

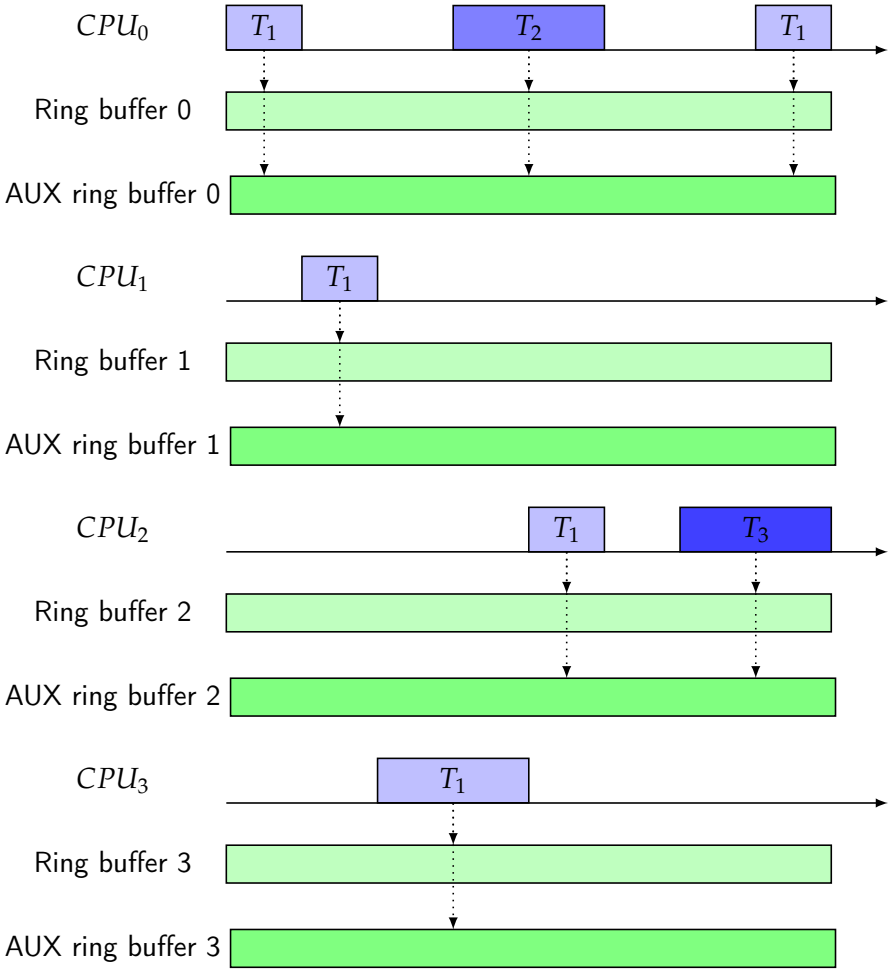

Figure 9: Ring buffers for system wide mode

AUX events

Similiar to `perf_output_begin()` and `perf_output_end()` for the generic ring buffer, the functions `perf_aux_output_begin()` and `perf_aux_output_end()` works as AUX ring buffer variant for processing the hardware trace data.

Same as the generic ring bufer, the structure `perf_output_handle` is used as a context to track the AUX buffer's info. Unlike `perf_output_begin()` allocates the buffer for only one sample, `perf_aux_output_begin()` creates the context and allocate all available buffer for filling AUX trace data.

The function `perf_aux_output_end()` finishes two things:

- It fills an event `PERF_RECORD_AUX` into the generic ring buffer, this event delivers the information of the start address and data size for a chunk of hardware trace data has been stored into the AUX ring buffer;
- The driver has stored new trace data into the AUX ring buffer, the argument `size` indicates how many bytes have been consumed by the hardware tracing, thus `perf_aux_output_end()` updates the header pointer to reflect the latest buffer usage.

There have two occasions to wake up the perf tool for reading out AUX trace data. Except the event `PERF_RECORD_AUX` can wake up the perf process, if the AUX trace size beyonds the watermark, the kernel also wakes up the perf process to fetch AUX trace data.

Perf tool saves events continuously in data file, the event `PERF_RECORD_AUX` is saved same as any other events. But the event `PERF_RECORD_AUX` doesn't contain any AUX trace data, it just presents a single recording event occurring in the kernel, and if the AUX ring buffer is big enough, it's possible that the multiple `PERF_RECORD_AUX` events occured prior to wake up the perf tool to handle the AUX ring buffer.

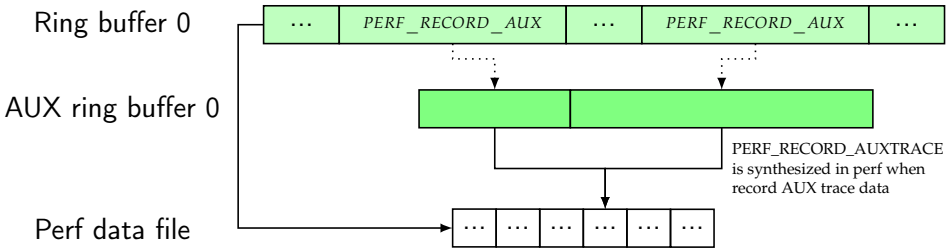

Figure 10: AUX events `PERF_RECORD_AUX` and `PERF_RECORD_AUXTRACE`

Perf needs to give out a clear boundary that "hey, this chunk in the perf file is the AUX trace data" when saves the AUX trace data into perf file and afterwards can be easily parsed in the post analysis. For this reason, perf tool synthesizes an event `PERF_RECORD_AUXTRACE` which combines the event header and AUX trace data.

Snapshot mode

Perf provides snapshot mode for AUX ring buffer, in this mode, users can only record AUX trace data at a specific time point which users are interested in. E.g. if an user wants to record the AUX trace data periodically with 1 second interval, it can use commands:

```
perf record -e cs_etm/@tmc_etr0/u -S -a program &
PERFPID=$!
while true; do
   kill -USR2 $PERFPID
   sleep 1
done
```

Let's firstly answer serveral questions to make clear the concept for snapshot: what's the snapshot and what's relationship between the snapsthot mode and the existed perf modes? The snapshot mode is only relevant to how to record the AUX tracing data, it is no matter with the ring buffers deployment. As we know, the buffers deployment is decided by per thread mode, per CPU mode or system wide mode, and the snapshot can be applied to any of these modes.

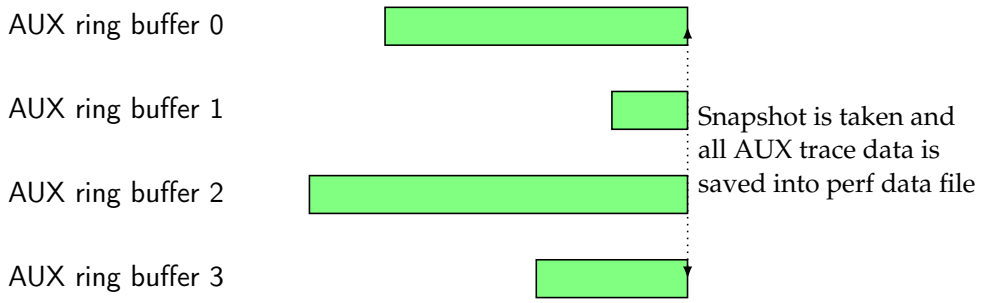

Figure 11: Snapshot with system wide mode

The main flow for snapshot is:

- Before the snapshot is taken, the AUX ring buffer runs in free mode; there have no the AUX event and the trace date will not be saved into the perf data file;
- When the user takes a snapshot for the profiling, it sends `USR2` signal to perf, perf will invoke the callback `auxtrace_record::snapshot_start()` to disable the hardware tracing; in the kernel the driver fills the hardware trace data into the AUX ring buffer and the event `PERF_RECORD_AUX` is saved in the generic ring buffer;
- `record__read_auxtrace_snapshot()` reads out the hardware trace data from AUX ring buffer, and saves into perf data file; during this process, the callback `auxtrace_record::find_snapshot()` is used to calculate the head and old head, these two heads will be described soon;
- After the snapshot is finished, `auxtrace_record::snapshot_finish()` will restart the PMU event for AUX tracing, so the hardware will generate new trace data.

A prominent difference for snapshot is it only uses the head pointer `aux_head` and doesn't update tail pointer `aux_tail`. The AUX ring buffer is possible that has not been overflow when snapshot, but if users take enough time for running the profiled program before taking snapshot, the AUX ring buffer can be filled with the AUX trace data for one or even multiple rounds,

therefore the tail pointer is useless and perf doesn't maintain it for snapshot mode. The AUX trace data size calculation is specific for AUX event, so the callback `auxtrace_record::find_snapshot()` is introduced, it needs to judge the head pointer has been wrapped around or not, and fixes up the head and the "old" head based on different situations.

Arm CoreSight: an example for using AUX ring buffer

This chapter reviews how Arm CoreSight uses AUX ring buffer (as writing this document the kernel's latest version is v5.13, all the discussion is based on the code in this version). It uses bottom-up approach to go through the Arm CoreSight implementation: it firstly summarizes the underlying hardware design, and then describes the implementation for how Arm CoreSight integrates the AUX ring buffer in details.

Recap for Arm CoreSight

The Arm architecture refers to a processor or core as a Processing Element (PE), any PE connected to a tracer can be recroded for its program flow by the tracer; the tracer is named as Embedded Trace Macrocell (ETM), nowdays the commonly used tracer IP is ETMv4 for Armv8 CPUs.

The data path is required for outputting trace data, a data path is established by links between tracers and sinks, a link can be a funnel or a replicator; a funnel is used to pass multiple tracers to a sink, and a replicator can be used to distribute the same source to different sinks.

The end point is a sink, different sinks are for different target medias. A sink can be Trace Port Interface Units (TPIUs) for dumping trace data to high speed port so the trace data can be captured by external debugger tool, Embedded Trace Buffers (ETB) for saving trace data to SRAM or Trace Memory Controllers (TMC) for accessing SDRAM.

Of course, the up description is highly condensed and it just wants to create a simplized hardware model to give readers a sense for how the CoreSight works with different components; the realistic hardware is much more complex.

Let's focus on the AUX ring buffer implementation, so we will take close look at TMC which uses the system memory for outputting trace data.

Three memory modes for Trace Memory Controllers (TMC)

TMC supports three memory modes: flat mode, scatter-gather mode, and Coresight Address Translation Unit (CATU) mode.

The memory modes can be divided into two kinds of pages allocation. The flat mode asks to allocate memory with physical continuous pages, TMC directly access the memory region based on the physical address. For scatter-gather mode and CATU mode, both supports the scatter-list pages, this gives the flexibility for using the discontinuous pages and relies on translation table for accessing physical pages.

Comparing against the scatter-gather mode, the CATU mode is improved for the physical address space length. The translation table entry is extended from scatter-gather mode's 32 bits to CATU mode's 64 bits, the scatter-gather mode only can support maximum to 40 bits of physical address space (the table entry's bit4-bit31 are mapped to physcial address bit12-bit39), in opposition CATU supports 64 bits of physical address space.

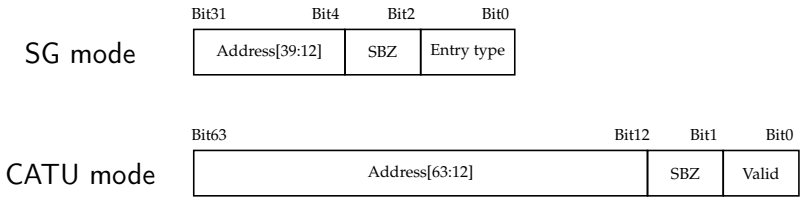

Figure 12: Table entry format for SG and CATU modes

Implementation AUX ring buffer for Arm CoreSight

Below four callbacks in the structure `pmu` play important roles for manipulating the AUX ring buffer.

```
int __init etm_perf_init(void)
{
        [...]

        etm_pmu.setup_aux           = etm_setup_aux;
        etm_pmu.free_aux            = etm_free_aux;
        etm_pmu.start               = etm_event_start;
        etm_pmu.stop                = etm_event_stop;

        [...]
}
```

As mentioned the AUX ring buffer has been allocated by `rb_alloc_aux()`, the original design anticipates calling `pmu::setup_aux()` to setup the PMU private data structures for the AUX ring buffer, e.g. creates the translation table for scatter-gather pages so that the sink can directly access the AUX ring buffer.

Unfortunately, Arm CoreSight's flat mode cannot support the scatter-gather pages, the solution is the function `etm_setup_aux()` invokes `sink_ops(sink)->alloc_buffer()`, the sink operation `sink_ops(sink)` is dynamically assigned based on the memory mode, whatever which mode is in use, the low level callback `sink_ops(sink)->alloc_buffer()` allocates a bounce buffer, at the end TMC rountes trace data into the bounce buffer.

It's easy to understand for applying the bounce buffer for flat mode; it's questionable for SG and CATU modes, which have already support scatter-gather list from the hardware design but why still use the bounce buffer in the software code? A reason is that this allows the implementation is general for all of these three modes, and the code is neat enough for later optimization.

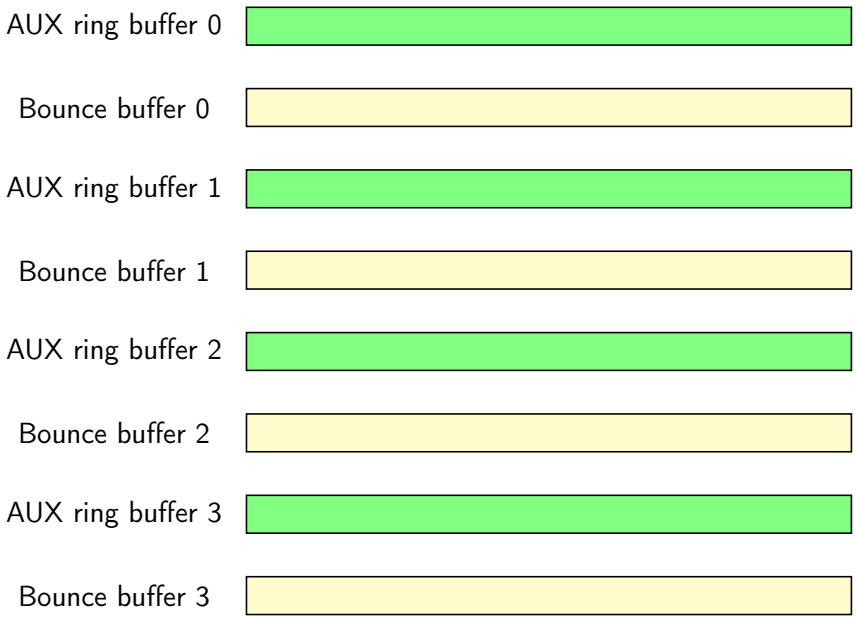

Figure 13: CoreSight bounce buffers for system wide mode

It is deserved to mention the memory mapping attribution for the bounce buffer. For the flat mode, the bounce buffer is allocated from DMA area with function `dma_alloc_coherent()`, the memory region is mapped with non-cacheable. This can cause performance downgrading when syncing data from the bounce buffer to the AUX ring buffer. For SG mode and CATU mode, they accesses the bounce buffer from the linear kernel mapping address, the memory mapping is normal attribution, so this is why before accessing the bounce buffer, it needs to call `dma_sync_single_for_cpu()` to sync the data writtend by device, this can avoid to read stale data in the cache and fetch the data from the memory.

The function `etm_event_start()` calls `perf_aux_output_start()` to prepare a context for AUX ring buffer, it enables the CoreSight components for the tracers, funnels and sinks. Usually, it establishes a path between ETM and ETR. If perf works as per CPU mode or system wide mode, there have multiple tracers should be enabled, the function `etm_event_start()` is called for multiple times, every calling enables a hardware path for a specific ETM.

When perf requires to record the AUX trace data, `etm_event_stop()` is executed. It stops the path by turning off tracer (ETM) and sink; when multiple tracers share the same one sink, it waits to copy AUX trace data from the bounce buffer to the AUX ring buffer until the last one tracer is disabled.

In theory, the bounce buffer and the AUX ring buffer should align with each other and keep the same pace for the trace data movement. But in reality, the bounce buffer is always free running, the AUX ring buffer needs to wait for the perf tool in user space to fetch data, it's possible the available free space in the AUX ring buffer is less than the new generated trace data in the bounce buffer; and the ETR driver can inject extra barrier packets as a boundary for easier decoding. These factors cause the misalignment between the bounce buffer and AUX ring buffer.

This is why the bounce buffer uses the structure `etr_buf` for managing the offset and size, based on the two values ETR driver can get to know what's the

source address and size for copying trace data. The copied size is capped by `perf_output_handle::size` which tracks how much the availible buffer size in AUX ring buffer. Rather than directly using `perf_output_handle::head` or the user page's `aux_head`, the ETR driver uses pointer `etr_perf_buffer::head` as the target address in the AUX ring buffer.

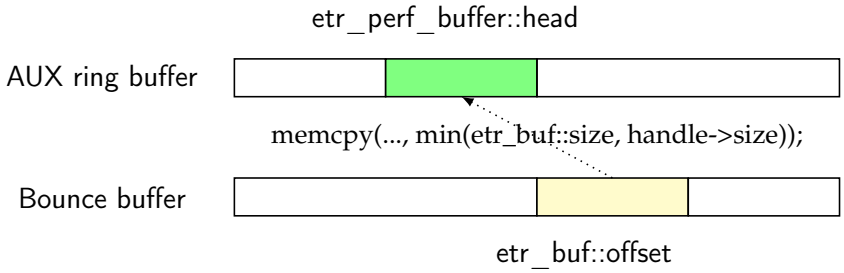

Figure 14: Syncing between bounce buffer and AUX ring buffer

At the end of perf session, `etm_free_aux()` frees the data structures and the bounce buffer.

Some thoughts for further optimization

As SG mode and CATU mode both support scatter-gather pages, the ETR driver can be improved to allow the hardware directly accessing AUX ring buffer; this can remove the redundant operations caused by the bounce buffer, the benefit is this can avoid the extra memory copying from the bounce buffer to the AUX buffer.

The second suggestion is to use normal memory mapping for the flat mode. Rather than using the DMA mapping, the normal mapping can utilize the cache for boosting memory performance. Though the memory size in flat mode is not so big with several megabytes, the performance improvement for memory copying with cacheable is still considerable. And given this overload is caused by the tool itself, the optimization can avoid the noise for the profiling.

A potential minor improvement is to remove `etr_perf_buffer::head`, this field can be replaced by using `perf_output_handle::head` which contains the header for AUX ring buffer.