BPF ring buffer

This document describes BPF ring buffer design, API, and implementation details.

Motivation

There are two distinctive motivators for this work, which are not satisfied by existing perf buffer, which prompted creation of a new ring buffer implementation.

- more efficient memory utilization by sharing ring buffer across CPUs;
- preserving ordering of events that happen sequentially in time, even across multiple CPUs (e.g., fork/exec/exit events for a task).

These two problems are independent, but perf buffer fails to satisfy both. Both are a result of a choice to have per-CPU perf ring buffer. Both can be also solved by having an MPSC implementation of ring buffer. The ordering problem could technically be solved for perf buffer with some in-kernel counting, but given the first one requires an MPSC buffer, the same solution would solve the second problem automatically.

Semantics and APIs

Single ring buffer is presented to BPF programs as an instance of BPF map of type `BPF_MAP_TYPE_RINGBUF`. Two other alternatives considered, but ultimately rejected.

One way would be to, similar to `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, make `BPF_MAP_TYPE_RINGBUF` could represent an array of ring buffers, but not enforce "same CPU only" rule. This would be more familiar interface compatible with existing perf buffer use in BPF, but would fail if application needed more advanced logic to lookup ring buffer by arbitrary key. `BPF_MAP_TYPE_HASH_OF_MAPS` addresses this with current approach. Additionally, given the performance of BPF ringbuf, many use cases would just opt into a simple single ring buffer shared among all CPUs, for which current approach would be an overkill.

Another approach could introduce a new concept, alongside BPF map, to represent generic "container" object, which doesn't necessarily have key/value interface with lookup/update/delete operations. This approach would add a lot of extra infrastructure that has to be built for observability and verifier support. It would also add another concept that BPF developers would have to familiarize themselves with, new syntax in libbpf, etc. But then would really provide no additional benefits over the approach of using a map. `BPF_MAP_TYPE_RINGBUF` doesn't support lookup/update/delete operations, but so doesn't few other map types (e.g., queue and stack; array doesn't support delete, etc).

The approach chosen has an advantage of re-using existing BPF map infrastructure (introspection APIs in kernel, libbpf support, etc), being familiar concept (no need to teach users a new type of object in BPF program), and utilizing existing tooling (bpftool). For common scenario of using a single ring buffer for all CPUs, it's as simple and straightforward, as would be with a dedicated "container" object. On the other hand, by being a map, it can be combined with `ARRAY_OF_MAPS` and `HASH_OF_MAPS` map-in-maps to implement a wide variety of topologies, from one ring buffer for each CPU (e.g., as a replacement for perf buffer use cases), to a complicated application hashing/sharding of ring buffers (e.g., having a small pool of ring buffers with hashed task's tgid being a look up key to preserve order, but reduce contention).

Key and value sizes are enforced to be zero. `max_entries` is used to specify the size of ring buffer and has to be a power of 2 value.

There are a bunch of similarities between perf buffer (`BPF_MAP_TYPE_PERF_EVENT_ARRAY`) and new BPF ring buffer semantics:

- variable-length records;
- if there is no more space left in ring buffer, reservation fails, no blocking;
- memory-mappable data area for user-space applications for ease of consumption and high performance;
- epoll notifications for new incoming data;
- but still the ability to do busy polling for new data to achieve the lowest latency, if necessary.

BPF ringbuf provides two sets of APIs to BPF programs:

- `bpf_ringbuf_output()` allows to *copy* data from one place to a ring buffer, similarly to `bpf_perf_event_output()`;
- `bpf_ringbuf_reserve()`/`bpf_ringbuf_commit()`/`bpf_ringbuf_discard()` APIs split the whole process into two steps. First, a fixed amount of space is reserved. If successful, a pointer to a data inside ring buffer data area is returned, which BPF programs can use similarly to a data inside array/hash maps. Once ready, this piece of memory is either committed or discarded. Discard is similar to commit, but makes consumer ignore the record.

`bpf_ringbuf_output()` has disadvantage of incurring extra memory copy, because record has to be prepared in some other place first. But it allows to submit records of the length that's not known to verifier beforehand. It also closely matches `bpf_perf_event_output()`, so will simplify migration significantly.

`bpf_ringbuf_reserve()` avoids the extra copy of memory by providing a memory pointer directly to ring buffer memory. In a lot of cases records are larger than BPF stack space allows, so many programs have use extra per-CPU array as a temporary heap for preparing sample. bpf_ringbuf_reserve() avoid this needs completely. But in exchange, it only allows a known constant size of memory to be reserved, such that verifier can verify that BPF program can't access memory outside its reserved record space. bpf_ringbuf_output(), while slightly slower due to extra memory copy, covers some use cases that are not suitable for `bpf_ringbuf_reserve()`.

The difference between commit and discard is very small. Discard just marks a record as discarded, and such records are supposed to be ignored by consumer code. Discard is useful for some advanced use-cases, such as ensuring all-or-nothing multi-record submission, or emulating temporary `malloc()`/`free()` within single BPF program invocation.

Each reserved record is tracked by verifier through existing reference-tracking logic, similar to socket ref-tracking. It is thus impossible to reserve a record, but forget to submit (or discard) it.

`bpf_ringbuf_query()` helper allows to query various properties of ring buffer. Currently 4 are supported:

- `BPF_RB_AVAIL_DATA` returns amount of unconsumed data in ring buffer;
- `BPF_RB_RING_SIZE` returns the size of ring buffer;
- `BPF_RB_CONS_POS`/`BPF_RB_PROD_POS` returns current logical possition of consumer/producer, respectively.

Returned values are momentarily snapshots of ring buffer state and could be off by the time helper returns, so this should be used only for debugging/reporting reasons or for implementing various heuristics, that take into account highly-changeable nature of some of those characteristics.

One such heuristic might involve more fine-grained control over poll/epoll notifications about new data availability in ring buffer. Together with `BPF_RB_NO_WAKEUP`/`BPF_RB_FORCE_WAKEUP` flags for output/commit/discard helpers, it allows BPF program a high degree of control and, e.g., more efficient batched notifications. Default self-balancing strategy, though, should be adequate for most applications and will work reliable and efficiently already.

Design and Implementation

This reserve/commit schema allows a natural way for multiple producers, either on different CPUs or even on the same CPU/in the same BPF program, to reserve independent records and work with them without blocking other producers. This means that if BPF program was interruped by another BPF program sharing the same ring buffer, they will both get a record reserved (provided there is enough space left) and can work with it and submit it independently. This applies to NMI context as well, except that due to using a spinlock during reservation, in NMI context, `bpf_ringbuf_reserve()` might fail to get a lock, in which case reservation will fail even if ring buffer is not full.

The ring buffer itself internally is implemented as a power-of-2 sized circular buffer, with two logical and ever-increasing counters (which might wrap around on 32-bit architectures, that's not a problem):

- consumer counter shows up to which logical position consumer consumed the data;
- producer counter denotes amount of data reserved by all producers.

Each time a record is reserved, producer that "owns" the record will successfully advance producer counter. At that point, data is still not yet ready to be consumed, though. Each record has 8 byte header, which contains the length of reserved record, as well as two extra bits: busy bit to denote that record is still being worked on, and discard bit, which might be set at commit time if record is discarded. In the latter case, consumer is supposed to skip the record and move on to the next one. Record header also encodes record's relative offset from the beginning of ring buffer data area (in pages). This allows `bpf_ringbuf_commit()`/`bpf_ringbuf_discard()` to accept only the pointer to the record itself, without requiring also the pointer to ring buffer itself. Ring buffer memory location will be restored from record metadata header. This significantly simplifies verifier, as well as improving API usability.

Producer counter increments are serialized under spinlock, so there is a strict ordering between reservations. Commits, on the other hand, are completely lockless and independent. All records become available to consumer in the order of reservations, but only after all previous records where already committed. It is thus possible for slow producers to temporarily hold off submitted records, that were reserved later.

One interesting implementation bit, that significantly simplifies (and thus speeds up as well) implementation of both producers and consumers is how data area is mapped twice contiguously back-to-back in the virtual memory. This allows to not take any special measures for samples that have to wrap around at the end of the circular buffer data area, because the next page after the last data page would be first data page again, and thus the sample will still appear completely contiguous in virtual memory. See comment and a simple ASCII diagram showing this visually in `bpf_ringbuf_area_alloc()`.

Another feature that distinguishes BPF ringbuf from perf ring buffer is a self-pacing notifications of new data being availability. `bpf_ringbuf_commit()` implementation will send a notification of new record being available after commit only if consumer has already caught up right up to the record being committed. If not, consumer still has to catch up and thus will see new data anyways without needing an extra poll notification. Benchmarks (see tools/testing/selftests/bpf/benchs/bench_ringbufs.c) show that this allows to achieve a very high throughput without having to resort to tricks like "notify only every Nth sample", which are necessary with perf buffer. For extreme cases, when BPF program wants more manual control of notifications, commit/discard/output helpers accept `BPF_RB_NO_WAKEUP` and `BPF_RB_FORCE_WAKEUP` flags, which give full control over notifications of data availability, but require extra caution and diligence in using this API.