Devlink DPIPE

Background

While performing the hardware offloading process, much of the hardware specifics cannot be presented. These details are useful for debugging, and `devlink-dpipe` provides a standardized way to provide visibility into the offloading process.

For example, the routing longest prefix match (LPM) algorithm used by the Linux kernel may differ from the hardware implementation. The pipeline debug API (DPIPE) is aimed at providing the user visibility into the ASIC's pipeline in a generic way.

The hardware offload process is expected to be done in a way that the user should not be able to distinguish between the hardware vs. software implementation. In this process, hardware specifics are neglected. In reality those details can have lots of meaning and should be exposed in some standard way.

This problem is made even more complex when one wishes to offload the control path of the whole networking stack to a switch ASIC. Due to differences in the hardware and software models some processes cannot be represented correctly.

One example is the kernel's LPM algorithm which in many cases differs greatly to the hardware implementation. The configuration API is the same, but one cannot rely on the Forward Information Base (FIB) to look like the Level Path Compression trie (LPC-trie) in hardware.

In many situations trying to analyze systems failure solely based on the kernel's dump may not be enough. By combining this data with complementary information about the underlying hardware, this debugging can be made easier; additionally, the information can be useful when debugging performance issues.

Overview

The `devlink-dpipe` interface closes this gap. The hardware's pipeline is modeled as a graph of match/action tables. Each table represents a specific hardware block. This model is not new, first being used by the P4 language.

Traditionally it has been used as an alternative model for hardware configuration, but the `devlink-dpipe` interface uses it for visibility purposes as a standard complementary tool. The system's view from `devlink-dpipe` should change according to the changes done by the standard configuration tools.

For example, itâ€™s quite common to implement Access Control Lists (ACL) using Ternary Content Addressable Memory (TCAM). The TCAM memory can be divided into TCAM regions. Complex TC filters can have multiple rules with different priorities and different lookup keys. On the other hand hardware TCAM regions have a predefined lookup key. Offloading the TC filter rules using TCAM engine can result in multiple TCAM regions being interconnected in a chain (which may affect the data path latency). In response to a new TC filter new tables should be created describing those regions.

Model

The `DPIPE` model introduces several objects:

- headers
- tables
- entries

A `header` describes packet formats and provides names for fields within the packet. A `table` describes hardware blocks. An `entry` describes the actual content of a specific table.

The hardware pipeline is not port specific, but rather describes the whole ASIC. Thus it is tied to the top of the `devlink` infrastructure.

Drivers can register and unregister tables at run time, in order to support dynamic behavior. This dynamic behavior is mandatory for describing hardware blocks like TCAM regions which can be allocated and freed dynamically.

`devlink-dpipe` generally is not intended for configuration. The exception is hardware counting for a specific table.

The following commands are used to obtain the `dpipe` objects from userspace:

- `table_get`: Receive a table's description.
- `headers_get`: Receive a device's supported headers.
- `entries_get`: Receive a table's current entries.
- `counters_set`: Enable or disable counters on a table.

Table

The driver should implement the following operations for each table:

- `matches_dump`: Dump the supported matches.
- `actions_dump`: Dump the supported actions.
- `entries_dump`: Dump the actual content of the table.
- `counters_set_update`: Synchronize hardware with counters enabled or disabled.

Header/Field

In a similar way to P4 headers and fields are used to describe a table's behavior. There is a slight difference between the standard protocol headers and specific ASIC metadata. The protocol headers should be declared in the `devlink` core API. On the other hand ASIC meta data is driver specific and should be defined in the driver. Additionally, each driver-specific devlink documentation file should document the driver-specific `dpipe` headers it implements. The headers and fields are identified by enumeration.

In order to provide further visibility some ASIC metadata fields could be mapped to kernel objects. For example, internal router interface indexes can be directly mapped to the net device ifindex. FIB table indexes used by different Virtual Routing and Forwarding (VRF) tables can be mapped to internal routing table indexes.

Match

Matches are kept primitive and close to hardware operation. Match types like LPM are not supported due to the fact that this is exactly a process we wish to describe in full detail. Example of matches:

- `field_exact`: Exact match on a specific field.
- `field_exact_mask`: Exact match on a specific field after masking.
- `field_range`: Match on a specific range.

The id's of the header and the field should be specified in order to identify the specific field. Furthermore, the header index should be specified in order to distinguish multiple headers of the same type in a packet (tunneling).

Action

Similar to match, the actions are kept primitive and close to hardware operation. For example:

- `field_modify`: Modify the field value.
- `field_inc`: Increment the field value.
- `push_header`: Add a header.
- `pop_header`: Remove a header.

Entry

Entries of a specific table can be dumped on demand. Each eentry is identified with an index and its properties are described by a list of match/action values and specific counter. By dumping the tables content the interactions between tables can be resolved.

Abstraction Example

The following is an example of the abstraction model of the L3 part of Mellanox Spectrum ASIC. The blocks are described in the order they appear in the pipeline. The table sizes in the following examples are not real hardware sizes and are provided for demonstration purposes.

LPM

The LPM algorithm can be implemented as a list of hash tables. Each hash table contains routes with the same prefix length. The root of the list is /32, and in case of a miss the hardware will continue to the next hash table. The depth of the search will affect the data path latency.

In case of a hit the entry contains information about the next stage of the pipeline which resolves the MAC address. The next stage can be either local host table for directly connected routes, or adjacency table for next-hops. The `meta.lpm_prefix` field is used to connect two LPM tables.

```
table lpm_prefix_16 {
  size: 4096,
  counters_enabled: true,
  match: { meta.vr_id: exact,
           ipv4.dst_addr: exact_mask,
           ipv6.dst_addr: exact_mask,
           meta.lpm_prefix: exact },
  action: { meta.adj_index: set,
            meta.adj_group_size: set,
            meta.rif_port: set,
            meta.lpm_prefix: set },
}
```

Local Host

In the case of local routes the LPM lookup already resolves the egress router interface (RIF), yet the exact MAC address is not known. The local host table is a hash table combining the output interface id with destination IP address as a key. The result is the MAC address.

```
table local_host {
  size: 4096,
  counters_enabled: true,
  match: { meta.rif_port: exact,
           ipv4.dst_addr: exact},
  action: { ethernet.daddr: set }
}
```

Adjacency

In case of remote routes this table does the ECMP. The LPM lookup results in ECMP group size and index that serves as a global offset into this table. Concurrently a hash of the packet is generated. Based on the ECMP group size and the packet's hash a local offset is generated. Multiple LPM entries can point to the same adjacency group.

```
table adjacency {
  size: 4096,
  counters_enabled: true,
  match: { meta.adj_index: exact,
           meta.adj_group_size: exact,
           meta.packet_hash_index: exact },
  action: { ethernet.daddr: set,
            meta.erif: set }
}
```

ERIF

In case the egress RIF and destination MAC have been resolved by previous tables this table does multiple operations like TTL decrease and MTU check. Then the decision of forward/drop is taken and the port L3 statistics are updated based on the packet's type (broadcast, unicast, multicast).

```
table erif {
  size: 800,
  counters_enabled: true,
  match: { meta.rif_port: exact,
           meta.is_l3_unicast: exact,
           meta.is_l3_broadcast: exact,
           meta.is_l3_multicast, exact },
  action: { meta.l3_drop: set,
            meta.l3_forward: set }
}
```