WHITE PAPER

Communications Service Providers
Scalable Packet Processing Frameworks

Queue Management and Load Balancing on Intel® Architecture

Telecommunications traffic bandwidth demands continue to rise. New approaches are needed to optimize deployments. Intel® Architecture systems, such as those found in a virtual network function, can transition to the Intel® Dynamic Load Balancer (Intel® DLB) to meet the demand.

Authors

Niall McDonnell
Principal Engineer

Gage Eads
Network Software Engineer

Introduction

As telecommunications traffic bandwidth demand continues to rise dramatically, and more sophisticated processing of this traffic is required by service providers, network functions are becoming more complex. This increased load means that Intel® architecture based implementations, such as those found in a virtual network function, typically require multiple processor cores, and new approaches are required to optimize such deployments.

This paper is for customers of multi-core Intel® architecture products who use software queuing structures extensively. Topics include traffic trends, along with the requirements and limitations of software queuing strategies for load balancing and traffic distribution.

Intel proudly introduces the Intel® Dynamic Load Balancer (Intel® DLB) solution. This white paper discusses Intel DLB features, goals, advantages and future product intercepts. Included is a summary of how Intel DLB is enabled with software and aligned with Data Plane Development Kit (DPDK) and Network Function Virtualization (NFV) readiness, providing insight on how DPDK customers can transition to Intel DLB.

Core-to-Core Communications in Software

The number of CPU cores on a modern processor has increased to double figures, and commonplace scenarios require cores to communicate with each other. Software-managed queues (or rings) in shared memory are a standard solution. See **Figure 1**. Queues connect a producer/consumer pair so that the producer inserts new elements at the *tail* of the ring, while the consumer removes older elements from the *head* of the ring. Each ring element typically references an *event*, such as a network packet or some other construct. The head and tail pointers are shared variables between producer and consumer—the producer modifies the tail only but must read the head to verify that the ring is not full, while the consumer does the converse.

Table of Contents

Memory-based Queuing in Intel® Architecture Systems

Modifying a pointer involves getting an exclusive copy in local core cache. The difficulty occurs when this has to be shared with the queue partner. Modern CPU cores are extremely performant when operating from their local cache, which can have read latencies in the 3 ns to 7 ns range. But the latency to fetch (or invalidate) the head/tail pointer from the local cache of a different core is in the 50 ns range, and the CPUs are not designed to handle this sort of latency without incurring stalls.

The latency issue affects pointers for both producer and consumer—each must get a shared copy of a line the other is modifying, and an exclusive copy of a line the other is reading. The case where a single queue connects two entities in this manner is referred to as single producer/single consumer (SP/SC), and is generally manageable in software. Any performance loss is typically mitigated using schemes to keep local copies of the shared pointers. But these schemes also have some drawbacks, including extra branching. Batching at either end also reduces cost-per-event, but it increases latency. Therefore, prefetching is not effective, since the queuing partner can share and access data at any moment, unlike other scenarios where latency is a known factor (for example, when fetching blocks of data from high-latency storage).

The case where multiple producers are involved is referred to as multi producer/single consumer (MP/SC). In this case, latency implications are more severe. Individual producers must update the tail pointer in an atomic manner to ensure consistency by locking the corresponding cache line, which incurs a cycle cost. Replacing a MP/SC queue with multiple SP/SC queues makes things simpler for the producers, but the burden is simply shifted to the consumer that must instead poll and manage multiple queue heads.

Cases with multiple consumers (MP/MC) impose the same locking requirements for the head pointer. These cases are even more complex if task order must be maintained.

As the number of cores/die grows, the cross-core latencies trend upwards relative to local cache latencies, and the likelihood of lock contention increases. The actual impact a software thread will see depends on the activity at the other end of the queue and, possibly, cache occupancy. As such, software-based queuing tends not to have very deterministic performance especially if multiple producers/consumers are involved.

Work Distribution

In many applications running on modern processors with a large number of cores, workloads must be distributed across a number of such cores. Consider packet processing as an example in this regard. As traffic bandwidth increases at a higher rate than compute availability, a good work distribution scheme is essential to optimize the available compute resources.

In packet processing, streams of incoming packets can exceed the capacity of any single core. So they are divided between available workers. Each packet stream contains individual flows, whose number and bandwidth varies and rapidly changes. Software must access and modify data structures unique to a flow in order to process the packets common to that flow. It is typically required that order should be maintained per flow (though not between individual flows) on an end-to-end basis.

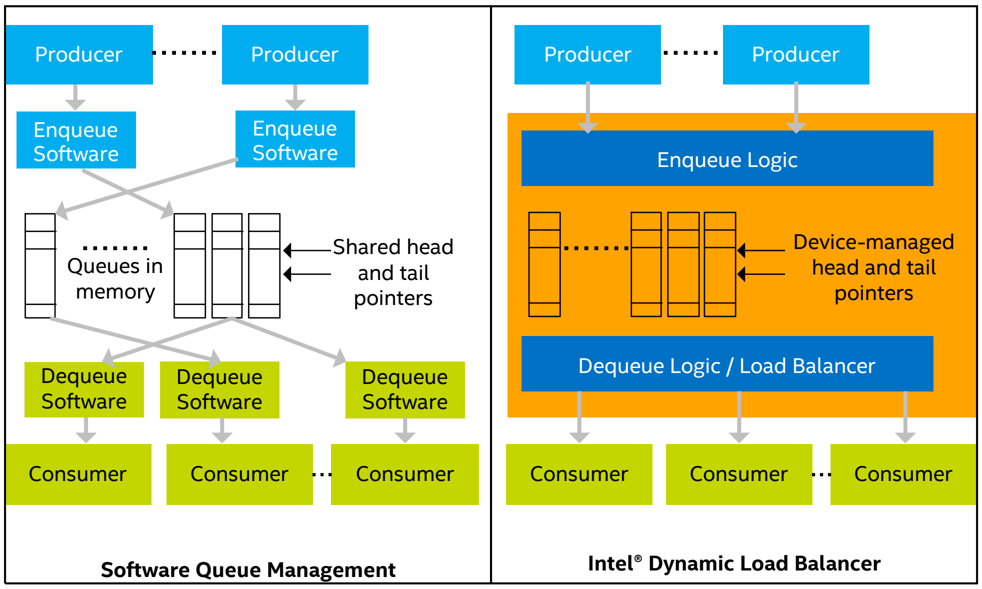

Figure 1. *Work Distribution Before and After Intel DLB*

There are two ways to split the workload:

- Pipelining breaks the processing flow into stages and places distinct stages on separate cores in a daisy chain fashion. Unfortunately, not all packets have the same processing flow, and flows are not typically decomposed easily into stages of equal size.
- Distribution/load balancing is a parallel approach where packets are sprayed across multiple workers that may be executing the same processing stage.

Many systems employ a hybrid approach whereby each packet encounters multiple pipelined stages with distribution across multiple workers at each individual stage.

Static Versus Dynamic Distribution

At this point it is useful to distinguish between *static sharing* and *dynamic load balancing*:

- In *static sharing*, a producer of work may share work across multiple consumers (i.e., workers) without regard to the state or occupancy of the individual workers. For example, the producer may statically assign individual flows to cores based on some flow identifier. An example of this is the receive side scaling (RSS) scheme employed by modern network interface cards (NICs). Such schemes are simple to implement but have several drawbacks:
  - They cannot guarantee that worker cores are equally busy, particularly if the number of significant flows is low.
  - The largest flow that can be handled is limited to the capacity of a single core.
  - If packet compute time varies, it is difficult to prevent packets getting stuck behind a burst of other traffic and suffering significant latency penalties. Prioritization can help but even high-priority traffic can get delayed behind traffic of a similar level.
  - The scheme is only as good as its least performant worker. This can be difficult to predict. Some cores may be subject to more interruptions than others or simply poorer caching effects.
  - It is necessary to keep all workers available in such schemes. They cannot reliably sleep for long durations, even with low traffic rates. The next arriving packet could be destined for any worker, unless there is a moderately complex work handoff scheme implemented.
- In *dynamic load balancing*, the producer attempts to distribute work with regard to the state/occupancy of the workers in an effort to ensure that they are approximately equally busy and used to their fullest extent. High-bandwidth flows are spread across multiple cores—but often, the original flow order must be restored thereafter. While this type of scheme overcomes many of the above drawbacks, it is more complex to implement. For example, to optimize power if traffic is below the provisioned rates, some cores can be taken offline to low power states. See Power-Aware Load Balancing on page 4.

As traffic bandwidth continues to increase, a good load balancing system is an essential requirement for packet processing.

Types of Load Balancing

In a packet-processing flow context, there are three types of distributions.

1. *Unordered* distribution sprays the packets across multiple worker cores. Software alone is not assumed to preserve the flow order when packets from the same flow go to different workers. Furthermore, multiple packets from the same flow may be outstanding on different workers simultaneously. This may require expensive synchronization mechanisms in the software. This type of processing is really only useful if there is no requirement to preserve order within a flow.
2. *Ordered* distribution is similar to unordered, except that the system provides a means of restoring the original flow order. Synchronization mechanisms may still be required in the software. This type of processing is useful if the bandwidth of individual flows approaches or exceeds the capability of individual cores. For use cases where the application is stateless, ordered distribution can achieve the best load balancing and performance.
3. *Atomic* distribution ensures that packets from a given flow can only be outstanding on a single core at a given time. It dynamically pins flows to cores, migrating flows between cores to load balance when required. This preserves flow order and allows the processing software to operate in a lock-free manner. As such, this type of distribution is highly desirable in modern packet processing equipment.

Implementations of the atomic and ordered distribution schemes require close cooperation between the producer and the workers/consumers. In schemes implemented in software, this sort of communication is implemented using multiple queuing structures and becomes subject to the performance limitations and non-determinism described above.

Current Performance of Load Balancing in Software

DPDK is an open source community based on software libraries originally developed by Intel. DPDK libraries include performance-optimized implementations of software queue management and work distribution. These libraries substantially reduce the cost of operations and successfully enable Intel® architecture customers. But with increased cores, the burden of software queue management increases accordingly. And since throughput rates increase faster than available compute, the cycles/packet budget shrinks. Because of these two factors, queue management is once again becoming a relatively expensive and performance-limiting operation.

On current CPUs, the performance of atomic load balancing is in the region of 15 M to 30 M decisions per second. Packet reordering cost is similarly expensive. CPU instructions-per-cycle improvements and batching (if the application allows) can lift this performance. But the basic limitation of cross-core latency will continue to increase with the number of cores/CPU.

Packet throughput expected from a system is increasing far more rapidly than core count; therefore, dedicating cores for load balancing is not sustainable. The performance impact is not limited to the cores doing the distribution work. All worker cores incur queue management costs in communicating with the distributor software and this can substantially affect worker performance.

Intel® Dynamic Load Balancer (Intel® DLB)

Introduction

The Intel DLB is a hardware managed system of queues and arbiters connecting producers and consumers. It is a PCI device in the CPU package. Intel DLP interacts with software running on cores and potentially other devices. Intel DLB implements the load balancing features outlined earlier, including the following:

- Lock-free multi-producer/multi-consumer operation.
- Multiple priorities for varying traffic types.
- Various distribution schemes.

Data-plane software communicates with Intel DLB using standard (PCI) memory mapped interfaces in a simple, low cycle-cost way that is enabled with DPDK.

Intel DLB supports virtualization using industry-standard techniques, and is exposed as part of the Virtual Network Function Infrastructure on an Intel® architecture platform. Intel DLB further allows finer grained isolation between individual applications if necessary.

Basic Intel DLB Operation

Intel DLB operates with the concept of resources, of which there are several kinds:

- *Ports* are memory mappable areas that enqueue to, or dequeue from, Intel DLB.
- *Queue IDs* (QIDs) are internal queues within the Intel DLB itself. A QID is a logical destination for a stream of packets that may be distributed across a number of workers according to each packet's load-balanced scheduling type. A QID maintains atomic and ordered distribution packets in order, at least on a per-flow and per-priority basis. The application specifies the QID it wishes to send a packet to at enqueue time.

Driver software allocates these resources to applications/VMs, which can in turn allocate to their individual threads as necessary. Properly configured, Intel DLB prevents cross application interference by discarding illegal traffic and ensuring each application cannot consume traffic to which it should not have access.

Performance

The rate at which Intel DLB distributes work determines overall system performance. The goal is to achieve a richer feature set and significantly greater performance than could be offered by software solutions.

Power-Aware Load Balancing

Intel DLB can rapidly vary the number of workers processing traffic dynamically according to traffic levels. Workers that are not in use can enter low-power states or can be made available for other tasks.

Enabling Intel DLB

Discovery and Enumeration

Within a running system, the Intel DLB instances, both physical function (PF) and virtual function (VF), are owned and controlled by a kernel driver. All device discovery and enumeration is handled by the kernel infrastructure for PCI devices. The kernel driver makes Intel DLB resources, such as ports, QIDs and credits available to applications in user space as they are requested by those applications.

DPDK Eventdev

DPDK offers a number of work distribution and load balancing schemes that can be used by applications. In many cases significant benefits can be realized by switching from using these existing schemes to using Intel DLB.

For data plane use, Intel DLB is enabled in DPDK as an instance of a class of a work/event scheduling device called an eventdev. This library was originally released in DPDK 17.05. The eventdev infrastructure is similar to that of the ethdev and cryptodev device types, in that a high-level API provides a common interface layer supported by individual drivers underneath it. Packets, or other events such as timer expiration, are enqueued by the application software to the eventdev device, which performs appropriate scheduling and prioritization. The scheduled events are retrieved by software when it calls the eventdev dequeue function. For the Intel DLB driver, each event to be scheduled corresponds to a queue entry (QE) inside the hardware.

Note

Intel DLB is not the only device supported under the eventdev device type. Ahead of the Intel DLB being generally available, multiple software implementations of an eventdev have been released, allowing applications to be developed and deployed on hardware without an Intel DLB. Those applications can then be transparently accelerated without any application code changes when deployed on a platform with an Intel DLB available.

For more details on the eventdev library, refer to the Event Device Library section of the DPDK programmer's guide[1].

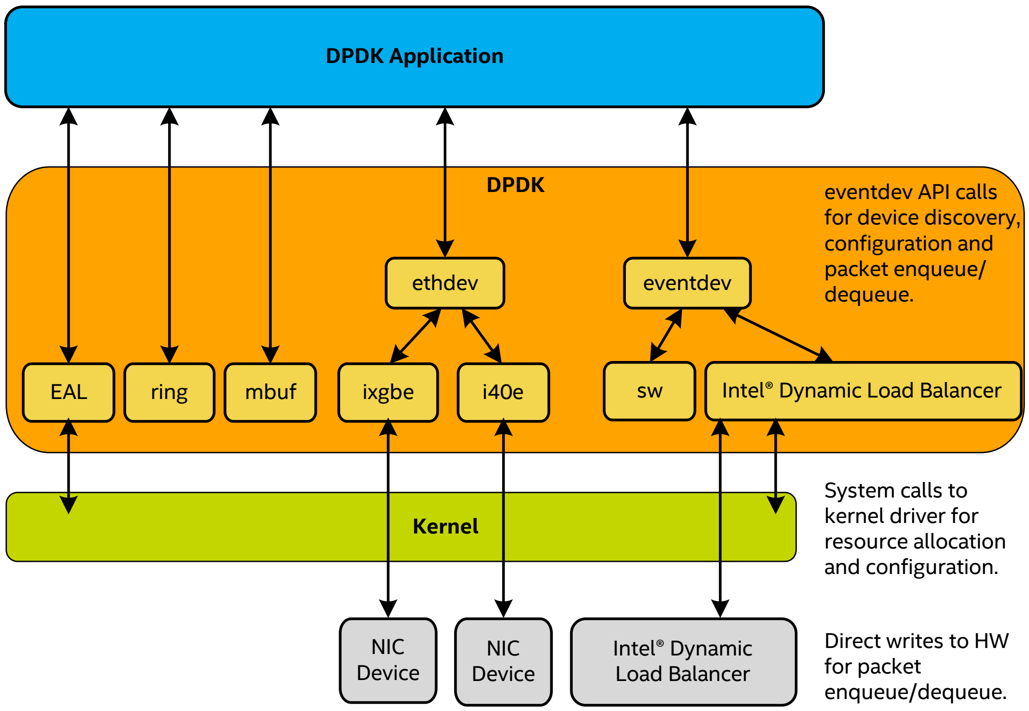
DPDK Application
DPDK
eventdev API calls for device discovery, configuration and packet enqueue/ dequeue.
ethdev
eventdev
EAL
ring
mbuf
ixgbe
i40e
sw
Intel® Dynamic Load Balancer
Kernel
System calls to kernel driver for resource allocation and configuration.
NIC Device
NIC Device
Intel® Dynamic Load Balancer
Direct writes to HW for packet enqueue/dequeue.

***Figure 2**. DPDK Application with Intel DLB*

Transitioning from Software Load Balancing to Intel DLB

DPDK includes the following main components:

- A software eventdev poll-mode driver (PMD) that mirrors the features of Intel DLB.
- A generic API for configuring and interfacing to an eventdev, with implementations for both the Intel DLB and software eventdev PMDs.

Prior to introducing eventdev in 2017, DPDK supported software load balancing through its packet distributor library[2]. This library dedicates one core to distribute packets to all other cores, which receive packets from the distributor and operate on them. Converting an application using the DPDK packet distributor to using eventdev does not involve significant reworking of the application packet handling code, as the basic principles of dynamic scheduling remain the same in both cases. To use Intel DLB, the initial setup code in the application configures the Intel DLB/eventdev queues and ports. But thereafter, the common runtime APIs, such as rte_distributor_get_pkt(), can be converted directly into equivalent eventdev APIs. After this is done, the application can be further changed, as needed, to take advantage of additional Intel DLB features.

DPDK eventdev Userspace API Overview

The eventdev device class was introduced in DPDK 17.05 after months of discussion and development by many stakeholders, including Intel. Although minor changes in the APIs may occur as contributors submit new drivers, the eventdev API is not expected to change significantly. eventdev drivers are already present, including a software implementation from Intel. This allows application development to be done using the eventdev API, the major functions of which are shown in **Figure 2**. Certain aspects of Intel DLB behavior, such as the credit scheme, may not be visible in the API as they are specific to the device.

The following APIs query event management sources.

```
rte_event_dev_count()         // Get number of event
                                 devices available
rte_event_dev_info_get()      // Get device
                                 parameters
```

The following APIs configure queues and ports.

```
rte_event_dev_configure()     // Configure an event
                                 device
rte_event_queue_setup()       // Allocate/configure a
                                 queue
rte_event_port_setup()        // Allocate/configure a
                                 port
rte_event_port_link()         // Map a queue to a
                                 port
```

The following APIs send or receive events.

```
rte_event_enqueue_burst()          // Burst enqueue
rte_event_dequeue_burst()          // Burst dequeue
```

For more detailed information on the eventdev API, refer to the eventdev API documentation[3].

Intel DLB Usage without DPDK

To support applications that do not wish to use DPDK, Intel DLB can also be enabled through a stand-alone client library named libdlb. While either eventdev or libdlb can be used to write Intel DLB-based applications, they differ in two key ways:

1. libdlb is independent of DPDK. This makes it appropriate for applications that can benefit from Intel DLB hardware but do not want to use the full DPDK framework. However, this means libdlb can only run on systems that have Intel DLB hardware; whereas eventdev applications are portable to a wide range of systems through the use of software-based event schedulers. libdlb could be extended with a software implementation in the future.
2. The libdlb API is tailored specifically to Intel DLB. It lacks the eventdev abstractions that are necessary to support a range of hardware and software event schedulers. Fewer abstractions generally leads to improved performance. This library also exposes a few Intel DLB concepts that do not exist in the eventdev API.

Optimizing Software Costs

An important aspect of any device like Intel DLB is the cost to software of interacting with the device. For dataplane software there are a few aspects to look at. The Intel DLB design, in conjunction with new ISA technologies, minimizes these costs resulting in a very efficient system.

IPsec Router Example

An IPsec router can be implemented using Intel DLB. Gateways are becoming increasingly complex but for simplicity this example considers VPN termination and clear text routing. Furthermore, only one traffic direction is considered, as shown in **Figure 3**.

The NIC can identify and filter IPsec packets and some may be capable of performing the decryption/authentication steps in advance of the packet reaching software. Otherwise, this step is performed in software or by using a look-aside accelerator. This example assumes that software (such as AES-NI) is used for the cryptographic operations.

One important parameter is the number and bandwidth of the IPsec tunnels. There are at least two possibilities here:

- In an enterprise application, the gateway may be terminating large numbers of VPNs of comparatively low bandwidth. The number of active tunnels is considered to be high and it may be feasible to use the NIC RSS scheme for initial per flow distribution, though it is still possible that some flows may have burst behavior which could cause load balancing issues.
- In a wireless edge application, the inverse is typically the case. The device is expected to terminate a comparatively small number of high bandwidth tunnels. RSS is not going to be very effective in this scenario.

Current Implementations

Several options are available in current Intel® architecture platforms. For the enterprise, RSS distribution is considered because there is no obvious need to incur the cost of core-to-core communications. Therefore, a run-to-completion model is assumed, where all processing for a given packet is executed on a single core.

There is no core-to-core communications penalty here and almost no locking is required in software, as long as there is no interdependency between IPsec tunnels. A possible exception is the NIC Tx stage, where a lock could be required for insertion into a Tx queue if such had to be shared.

This model has been successfully deployed, though it has drawbacks:

- RSS can still give imperfect distribution. Therefore, performance must be guard-banded, which limits overall guaranteed performance.
- More sophisticated gateways may require more advanced schemes. For example, it may be required to atomically process VPN traffic in both directions, which is not possible using this setup.
- It is not trivial to group disparate inner tunnels together for atomic processing such as may be required for traffic shaping.

The wireless use case is more difficult to address; RSS may be insufficient and the higher bandwidth tunnels may overwhelm a single core. This could necessitate splitting the processing pipeline across multiple cores in an SP/SC manner. But this will typically create considerable inequality in core utilization.

Intel DLB Based Implementation

In a system with Intel DLB, a number of options are available. In the simplest case, the NIC RSS scheme can be replaced by an ingress core running the poll mode driver to fetch Rx descriptors and then submitting the packets to Intel DLB for atomic load-balancing. The workers can then resubmit to Intel DLB for a direct processing stage to a dedicated Tx core.

The advantage of this over the software-only scheme is higher guaranteed performance due to better load balancing compared to RSS.

If the bandwidth of a single large IPsec tunnel could overwhelm a core, ordered distribution may be necessary. This typically results in a more complex approach with multiple passes through Intel DLB for each packet. The worker cores can be broken into one or more groups that may be active at differing stages on the pipeline.

In the example above, the first ordered stage allows a high bandwidth IPsec tunnel to be load-balanced across multiple cores. These packets are put back in their original order by Intel DLB on resubmission to the following stage. Subsequent stages are atomic, as it is assumed that locks are to be avoided, with the same direct stage to Intel DLB at the end.

This example shows how additional stages, with each stage essentially a QID, can be added for flow creation (setup of new table entries), traffic shaping, or other more complex features. In NFVs that provide more complex functions than routing, this is the expected usage.

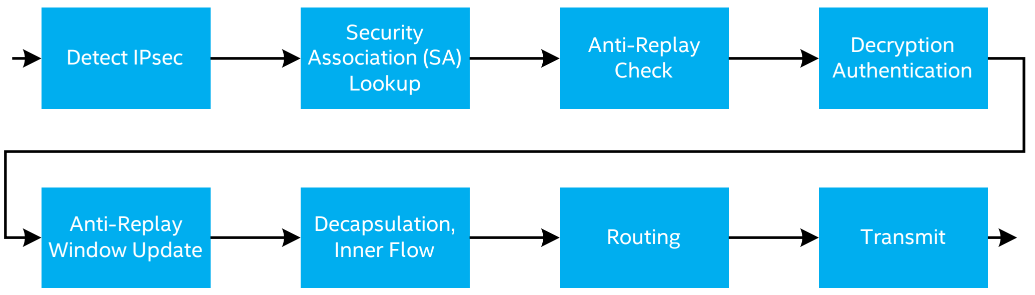
Detect IPsec
Security Association (SA) Lookup
Anti-Replay Check
Decryption Authentication
Anti-Replay Window Update
Decapsulation, Inner Flow
Routing
Transmit

***Figure 3**. IPsec Router Pipeline*

Summary

The Intel DLB is a decapsulation new device on Intel platforms. It offers new mechanisms for core-to-core communication with built-in load balancing capability. Advantages expected for packet processing applications using Intel DLB are as follows:

- Much higher load-balancing performance than existing software schemes
- Cost to worker cores lower than in software schemes
- Lock free multi-producer to single-consumer optimization
- Better determinism
- Built in priority
- Built in performance monitoring
- Better flexibility and granularity

Intel DLB can be enabled under DPDK APIs. DPDK customers should see a pain-free transition to new hardware from the software eventdev PMD.

[1] https://doc.dpdk.org/guides/prog_guide/eventdev.html
[2] https://doc.dpdk.org/guides/prog_guide/packet_distrib_lib.html
[3] https://doc.dpdk.org/api/rte__eventdev_8h.html