Technology Guide

Intel® Data Streaming Accelerator (DSA) - Packet Copy Offload in DPDK with Intel® DSA

Authors

Bruce Richardson

1 Introduction

The Data Plane Development Kit (DPDK), has become the de facto standard on which packet processing applications and other packet processing frameworks are based. While DPDK, and high-performance packet processing frameworks, have taken the concept of "zero-copy" packet processing to heart, there are scenarios in real-world applications where packet copies are necessary, and in these scenarios the computational cost of performing those copies can be significant, especially for larger packet sizes. This document describes how such packet copies can be offloaded to copy or Direct Memory Access (DMA) accelerators, such as the Intel® Data Streaming Accelerator, by way of the new DMA-offload library, "dmadev", introduced in the DPDK 21.11 release.

This document is part of the Network Transformation Experience Kits.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	December 2022	Initial release.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
DPDK	Data Plane Development Kit
DMA	Direct Memory Access
FD.io	Fast Data Input/Output
TCP	Transmission Control Protocol
LTS	Long Term Stable
NUMA	Non-Uniform Memory Access
Virtio	Virtualized Input/Output
VPP	Vector Packet Processing

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
DPDK Driver documentation	https://doc.dpdk.org/guides-22.11/dmadevs/idxd.html

2 Overview

In high-performance packet processing applications that use libraries and toolkits such as the DPDK or the VPP Framework from Fd.io project, the packet pipeline is designed to minimize or eliminate packet data copies. Where necessary, these copies can take up a significant number of processor cycles and can become significant performance bottlenecks in the application.

For example, in a TCP network stack or in a vSwitch application using virtio, these copy operations can similarly be seen as significant performance hotspots.

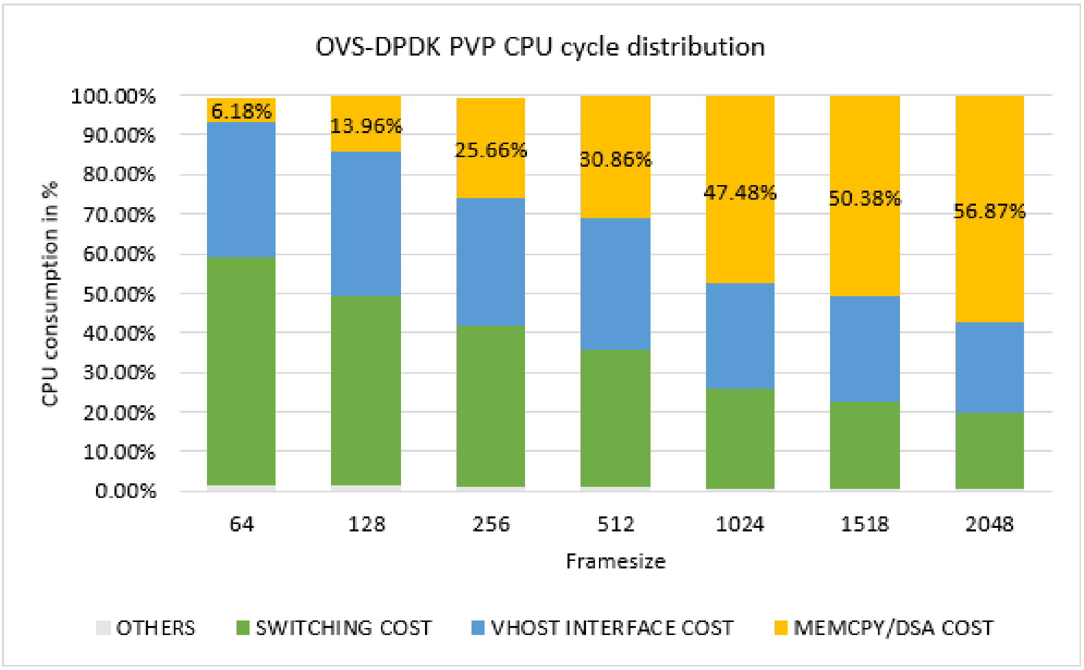

Figure 1 Relative cost of tasks in switching a packet from host to guest in a vswitch as packet sizes increase

3 Intel® Data Streaming Accelerator (Intel® DSA) Hardware Accelerator

Modern hardware platforms, such as those using the latest Intel® Xeon® processors, often include a variety of accelerators to increase performance of selected workloads. Copying data is no exception to this and DMA accelerator devices can be used on many platforms to have copy, fill, and other operations performed by the accelerator, thereby saving CPU core cycles for other parts of the application workload.

One such DMA acceleration device is the Intel® Data Streaming Accelerator (Intel® DSA), found on the 4th Gen Intel® Xeon® Scalable processors. This accelerator supports both streaming data movement, that is, copy, and data transformation operations, applicable to a wide variety of applications. Besides copy operations, Intel DSA supports other operations including:

- "Fill" operations, to place a repeating pattern into memory
- CRC generation, to checksum an area of memory
- Memory comparison operations
- Cache flushing

These operations are all presented to the user through a queue-based interface, where each hardware queue may be dedicated to a particular application or shared among multiple applications[1]. The configuration and assignment of these queues to applications is controlled by the Linux kernel driver for Intel DSA – called "idxd" – with the end-user configuring the device and its queues through applications like "accel-config", which communicate with the driver by writing to sysfs nodes.

Once configured, an application can take advantage of the accelerator either by directly interfacing with it through the Linux device nodes presented by the kernel driver, or more simply through higher level software such as the Intel® Data Mover Library (Intel® DML), or through the DPDK DMA device infrastructure, described in the next section. For DPDK applications, this latter DMA infrastructure should ensure cleanest integration and highest performance for DMA offload.

4 DPDK DMA Device Support

While the first releases of DPDK included only support for network adapters, in recent years DPDK has expanded to cover a much wider range of device types. The most recent LTS (Long Term Stable) release, DPDK 21.11, included support for:

- Baseband devices
- Compression devices
- Crypto Acceleration devices
- Event/Load-balancing devices
- General-Purpose Graphics Processing Unit (GPU) devices
- Network Adapters
- Regular Expression (Regex) Devices
- Virtio Data Plane Acceleration (vDPA) Devices
- "Raw" or uncategorized devices

As well as a newly added device category:

- DMA Acceleration Devices

This new "dmadev" library is the result of a multi-vendor collaboration within the DPDK open-source community, and, alongside the basic dmadev library, the 21.11 release includes DMA drivers from Intel, Marvell, HiSilicon, and NXP. The 21.11 release also includes a set of unit tests for dmadev, which can be used to verify that drivers correctly implement the APIs, and an example application demonstrating use of the APIs for copying packets in a packet processing application.

[1] For packet processing applications using DPDK, only dedicated queues are supported, ensuring that any applications are not contending with others for the queue resources.

4.1 Device Capabilities

As described in the public header "rte_dmadev.h"[2], the DMA framework is based upon the following model:

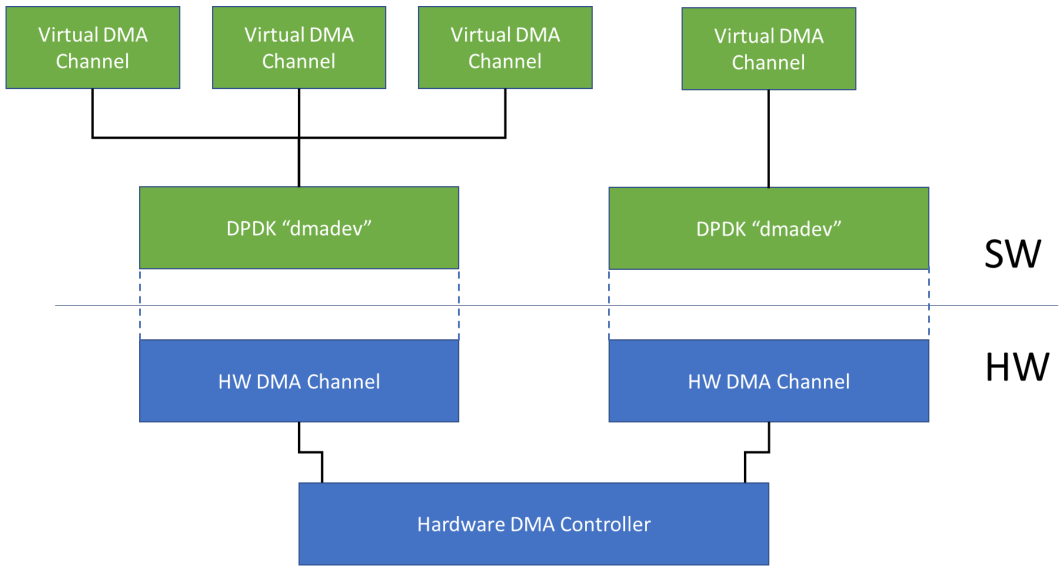

Figure 2. DPDK DMA Device Framework

Under this model, each hardware DMA channel is exposed to software as a single DMA device instance. That device instance can support one or more virtual channels (vchannels) on it, where each channel represents a different transfer context. For Intel DSA, each dmadev instance, that is, HW device queue, allows only a single vchannel, but all operations supported by the hardware can be sent on that channel. For other hardware, it may be necessary to create a new virtual channel for each operation type, and so multiple vchannels would be supported in those cases.

To account for the differences between the various hardware and driver implementations, the dmadev API includes the function "rte_dma_info_get()" to return details of a specific instance. As well as standard device information, such as the NUMA node the device is on, the returned structure provides details such as the number of vchannels a device supports, the maximum number of entries that can be used in a descriptor ring for a channel on that device, and a set of device capability flags. These capabilities provide extended information about what each device can do, and include details on whether each device supports:

- Copy, fill, or other operation types
- Copying to/from devices as well as to/from memory
- Additional capabilities; such as Shared Virtual Addressing (SVA) – where virtual addresses can be used instead of requiring IO addresses

While it is likely that all dmadev instances will support basic functionality such as memory-to-memory copies, all well-behaved applications should check the capabilities of each dmadev instance before using it.

4.2 Configuring Intel DSA Instance for DPDK Use

Intel DSA hardware can be configured for DPDK use in one of two different ways, shown in the diagram below. In the first instance, shown on the left, the Intel DSA device can be completely dedicated for DPDK use by binding it to the Linux kernel "vfio-pci" driver, or to a similar userspace IO driver on other operating systems. This is the same as what is done with many other devices and device virtual functions (VFs), such as Intel network cards. When bound in this way, on application initialization, DPDK will automatically configure all device resources on the Intel DSA instance, distributing them equally among all the available queues[3].

[2] https://doc.dpdk.org/api/rte__dmadev_8h.html

[3] If fewer queues are required than that provided by the DSA device, the number of configured queues may be limited by using the "max_queues" driver parameter as shown in the DPDK idxd driver documentation.

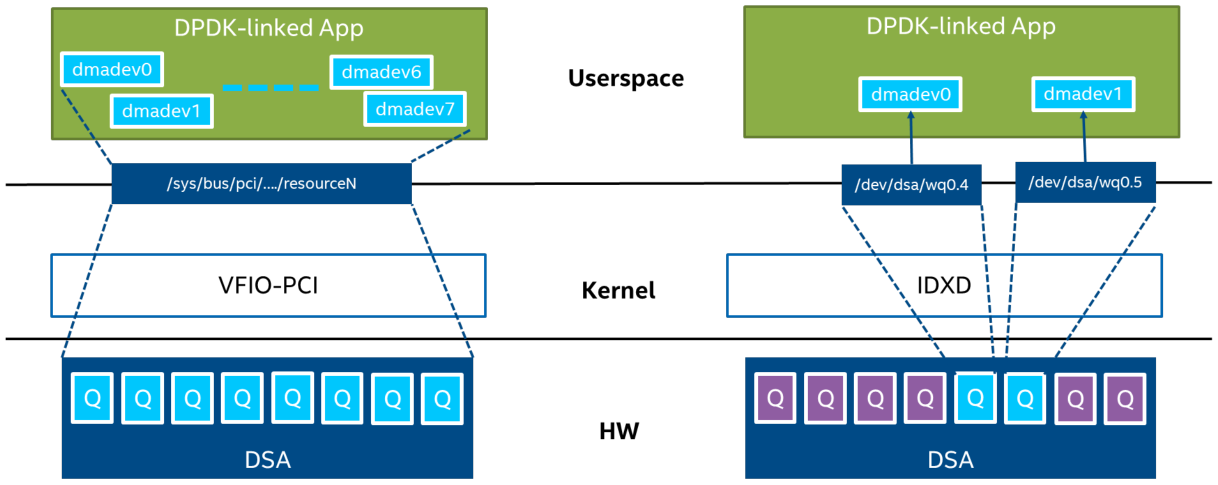

Figure 3. Two ways of mapping DSA resources into DPDK applications – taking a whole DSA instance via vfio-pci, or taking individual queue resources via the "idxd" Linux kernel driver.

The second way of configuring an Intel DSA device for DPDK use is to keep it bound to the Linux kernel "idxd" driver and configure specific queue resources on that device for DPDK use. This has the advantage that it allows a single Intel DSA instance to be shared among multiple DPDK processes or non-DPDK processes, for cases where a full device instance is not required.

To prepare specific Intel DSA queues for DPDK use, the queues must be configured as follows:

- Must be marked for userspace use
- Must be dedicated queues, rather than shared
- Must be assigned a name value starting with either "dpdk_" or, in the case of multiple DPDK processes running, the "file-prefix" value used when starting each process.

For example, the "accel-config" tool can be used to configure Intel DSA queues for DPDK using the commands below (taken from the DPDK driver documentation):

```
# configure 4 groups, each with one engine
accel-config config-engine dsa0/engine0.0 --group-id=0
accel-config config-engine dsa0/engine0.1 --group-id=1
accel-config config-engine dsa0/engine0.2 --group-id=2
accel-config config-engine dsa0/engine0.3 --group-id=3

# configure 4 queues, putting each in a different group, so each
# is backed by a single engine
accel-config config-wq dsa0/wq0.0 --group-id=0 --type=user --wq-size=32 \
    --priority=10 --max-batch-size=1024 --mode=dedicated --name=dpdk_app1
accel-config config-wq dsa0/wq0.1 --group-id=1 --type=user --wq-size=32 \
    --priority=10 --max-batch-size=1024 --mode=dedicated --name=dpdk_app1
accel-config config-wq dsa0/wq0.2 --group-id=2 --type=user --wq-size=32 \
    --priority=10 --max-batch-size=1024 --mode=dedicated --name=dpdk_app1
accel-config config-wq dsa0/wq0.3 --group-id=3 --type=user --wq-size=32 \
    --priority=10 --max-batch-size=1024 --mode=dedicated --name=dpdk_app1

# enable device and queues
accel-config enable-device dsa0
accel-config enable-wq dsa0/wq0.0 dsa0/wq0.1 dsa0/wq0.2 dsa0/wq0.3
```

4.3 Initializing a Device Instance

Each DMA device instance, or DSA queue configured for DPDK use, is found by DPDK as part of its normal PCI and other bus scanning process, and each device can be queried for its information and capabilities immediately after the call to "rte_eal_init()" to initialize DPDK.

Configuration of a DMA device is a relatively simple procedure, and an example can be seen in the "configure_dmadev_queue()" function in the "dma" example application in DPDK 21.11. This function, shown below, is also described the in the "Sample Applications Guide" for DPDK.

```
static void
configure_dmadev_queue(uint32_t dev_id)
{
        struct rte_dma_info info;
        struct rte_dma_conf dev_config = { .nb_vchans = 1 };
        struct rte_dma_vchan_conf qconf = {
                .direction = RTE_DMA_DIR_MEM_TO_MEM,
                .nb_desc = ring_size
        };
        uint16_t vchan = 0;

        if (rte_dma_configure(dev_id, &dev_config) != 0)
                rte_exit(EXIT_FAILURE, "Error with rte_dma_configure()\n");

        if (rte_dma_vchan_setup(dev_id, vchan, &qconf) != 0) {
                printf("Error with queue configuration\n");
                rte_panic();
        }
        rte_dma_info_get(dev_id, &info);
        if (info.nb_vchans != 1) {
                printf("Error, no configured queues reported on device id %u\n", dev_id);
                rte_panic();
        }
        if (rte_dma_start(dev_id) != 0)
                rte_exit(EXIT_FAILURE, "Error with rte_dma_start()\n");
}
```

The first step in device initialization is to call "configure" to set the number of virtual channels needed – for simple cases of memory-to-memory copies, this should generally only need to be "1". Thereafter, each vchannel, or queue, must be configured with operation types and queue descriptor ring sizes. In all cases, the parameters for number of vchannels and size of descriptor ring should be within the bounds reported by the "info_get" function for the device. [Not shown in example above].

Assuming no errors are encountered in configuring a device and its queue, the device can then be started using the appropriately named, "rte_dma_start()" function.

4.4 Submitting DMA Operations

While the DMA device library is designed to operate on bursts of packets, or jobs, at once; unlike many other libraries, such as the DPDK ethernet device (ethdev) library, the dmadev library does not provide explicit burst APIs. Instead, the library provides separate operations for 1) enqueuing operations one at a time, and then 2) submitting all those queued jobs as a single burst to the hardware[4].

Within an application, the processing flow for offloading copies to the hardware can therefore look something like the code below, where "srcs" and "dsts" are arrays of DPDK packet buffers, known as "mbufs":

```
        for (i = 0; i < nb_bufs; i++) {
                id = rte_dma_copy(dev_id, vchan_id,
                                rte_pktmbuf_iova(srcs[i]),
                                rte_pktmbuf_iova(dsts[i]),
                                srcs[i]->data_len);
                if (id < 0) {
                        /* error handling */
                }
                /* save any extra job state here */
        }

        rte_dma_submit(dev_id, vchan_id);
```

To enable tracking of the offloaded operations, the individual operation APIs, copy, fill, etc. return a job id, which is an incrementing value from 0 to UINT16_MAX (0xFFFF). This id can be masked to any power-of-2 size, allowing the application to manage any job state data in a local array, or arrays, of a suitable size. State data beyond the job id is not maintained by the library, allowing it to be smaller and more efficient and giving the application maximum flexibility in choosing the best storage layout for state data. An example of how this may be done can be seen in the "dma" example application included with DPDK 21.11.

4.5 Handling Job Completions

For best use of the DMA offload, the application should not wait for the operations to complete immediately after they have been submitted to hardware. Instead, the application should use the time while the operations are in flight to do other work, for example, prepare the next batch of packets for offloading. Later, the application should poll the DMA device for information on completed operations and continue the processing of the packets whose copies were offloaded.

In the simple case, applications can call the API "rte_dma_completed()" to return information about the jobs that are finished. This API will return N, the number of jobs completed, and, since jobs are never reported completed out-of-order, this information can be used to allow processing to continue on the next N packets that were offloaded. Alternatively, if the "last_idx" parameter is passed to the API, on function return, the ID of the last completed job will be filled into the variable pointed to by that parameter.

In some situations, it is possible that an error may occur during a copy operation – for example, if unpinned memory is used. While this should not normally occur in DPDK applications where all packet buffer memory is pinned and accessible for DMA by devices, the "has_error" parameter to the "rte_dma_completed()" API can report that an error occurred in processing. In case of such an error, the return value from the function is the number of successful operations, and the "rte_dma_completed_status()" API can be used to get more details of the error[5].

5 Summary

This paper describes how data copies can take up significant processing time inside some packet processing applications, such as virtual switches like Open vSwitch (OVS). The document also introduces the Intel® Data Streaming Accelerator hardware as an accelerator for offloading these packet copies to hardware. This offload is available to DPDK applications through the newly introduced "dmadev" API. The configuration and data-plane offload APIs are also discussed in detail showing how to use the dmadev APIs from end-user applications.

[4] Having a single enqueue API and a separate doorbell/submission API avoids the overhead of having the application build up an array of structures in memory/cache for passing into the burst function, where those structures then need to be re-read back in and converted to descriptors inside the driver. Instead, as the enqueue function is called for each operation in the burst, the parameters can be passed to the driver via registers, and the descriptor written directly to memory without having to go through any intermediate structures, and without using any extra memory reads and writes

[5] If error conditions are expected to occur regularly, the latter API can always be used for completion gathering rather than the simple "rte_dma_completed()" API.

1222/DN/WIT/PDF 758769-001US