Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 ECTS | Bachelor's Programme in Programming
202019 | LIU-IDA/LITH-EX-G--2019/034--SE

Improving the flexibility of DPDK Service Cores

Förbättring av flexibiliteten hos DPDK Service Cores

Denis Blazevic
Magnus Jansson

Supervisor : Rita Kovordanyi
Examiner : Jalal Maleki

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Improving the flexibility of DPDK Service Cores

Denis Blazevic
Linköping University,
Linköping, Sweden
denis.blaze@outlook.com

Magnus Jansson
Linköping University,
Linköping, Sweden
magjan@outlook.com

ABSTRACT
Data Plane Development Kit is a highly used library for creating network applications that can be run on all hardware. Data Plane Development Kit has a component called Service Cores, which allows the main applications to create services that will run independently. These services are manually mapped to specific CPU cores, and are scheduled in a round-robin method. Because of the manual mapping, and the scheduling, the different load for each service can impact the start time for each service. By having services not run when supposed to, the throughput will degrade. In this thesis, we investigate and try to solve the issue by implementing a basic load balancer into the Service Core component. Our results show that an basic load balancer, that will balance upon reaching a CPU upper threshold, will increase the throughput of services while decreasing the delay between each service run.

Author Keywords
Data Plane Development Kit; DPDK; network; networks; high-speed networking; Service Cores, Load Balancer, Multi-Core

1. INTRODUCTION

According to the Ericsson Mobility Report, the yearly growth in mobile data traffic has exceeded 50% during the last five years. This is driven by the increase of mobile data subscriptions and the data use per subscription. However, Ericsson also estimates an increase of three billion Internet of Things connections by the year 2024. This suggests a need for more bandwidth, and in many cases also lower latency. Production environments with moving robots is one of the examples where lower latency will be needed. These are some of the challenges that are addressed by the new 5G networks [1].

Due to operation management and the need for more flexibility, network operators want to move most of the network processing to a local cloud by virtualization. This means that instead of having dedicated hardware units (i.e. firewalls, routers, etc) these functions can be performed in software running on standard servers and switches. One building block for this transition is the Data Plane Development Kit library.

Data Plane Development Kit (DPDK)[1] is a library originally made by Intel, to make network packet processing faster, more power efficient and universal. From the beginning, it was made for the x86 architecture, but today, the library supports all major architectures and Network Interface Controller (NIC). DPDK operates in user-space but uses an abstraction layer so that applications do not have to be developed for a specific system. This abstraction makes it possible to write code that will work on any supported architecture.

Furthermore, even though DPDK resides in userspace it does not want the operating system scheduler to manage worker threads. The operating system scheduler uses interrupts to switch between running threads or run operating system functions. However, the use of interrupts would severely impact the performance of DPDK applications. Instead, DPDK worker threads want to run exclusively on a processor core and refrain from using interrupts. The consequence of this would mean that the use of interrupts will be exchanged for a polling method instead. In DPDK, the polling is done directly in an application or by using a poll method from a Poll Mode Driver.

The poll method allows packet processing applications to have multiple stages of processing. For example, the application could poll a number of packets from a receive queue, decrypt them and calculate where to forward this packet, encrypt it and then finally put it into the send data structure. These applications are usually run in one of the following two modes, *run to completion mode* or *pipeline mode*. However, since the introduction of Service Cores, the multiple stages of these applications can instead be run as services on specific CPU cores.

Service cores is an alternative approach to abstracting underlying hardware. Some hardware platforms might have a specific function implemented in hardware but other platforms might not. Service Cores abstracts away the system differences by allowing developers to implement services that can fall back to a software implementation. For example, some stages of packet processing applications might require functionality such as cryptography, which can be implemented in hardware. However, if a system lacks the necessary components for this, the Service Core library can be used to create services that will fall back to a software implementation if needed. Functionality such as cryptography could then be done as a service. But this would make the service require CPU resources [2].

Services are scheduled in a simple round-robin run-to-completion manner. This means that a service will have to finish its work before the next service will run. However, some services will need more CPU cycles than others. Because of this, some services will have to wait for a longer period to get the possibility to run. This can lead to higher

[1] https://www.dpdk.org/about/

[2] https://doc.dpdk.org/guides/prog_guide/service_cores.html

latency and lower throughput. This disadvantage has lead to our research questions.

1.1 Research Questions

In this paper, we are going to investigate how Data Plane Development Kit (DPDK) could be extended to support load balancing for services running on service cores with the aim to minimize waiting times and maximize throughput.

- RQ1: Could the use of a load balancer improve the resource efficiency by dynamically distributing services over service cores?
- RQ2: How would the latency and throughput, compared to a standard application, be affected by the load balancer?

Since a lot of computing and network resources are used by people it is not surprising that network traffic changes during the day and night. One challenge is to have enough network capacity to handle peak hours and then be able to scale down to conserve energy. A load balancer for packet processing should be able to distribute the workload across all available cores rapidly and when the workload drops it should gradually scale down the services to run on fewer cores without hurting throughput.

Using the load balancer, a comparison of exact same applications will be done but on two different systems - one with the load balancer turned off, and the second with the load balancer turned on. These systems run DPDK 19.05.rc1 on the Ubuntu 18.04.2 LTS distribution. Can a significant difference be seen?

2. THEORY

To understand why the Service Core library was implemented into DPDK, one needs to first understand how DPDK works.

2.1 Data Plane Development Kit

DPDK contains a number of libraries with the purpose of abstracting the underlying hardware and operating system. Additionally, DPDK has some specific features to support the making of packet processing applications.

2.1.1 Environment Abstraction Layer

To be able to use DPDK on a variety of hardware, an abstraction layer is needed. This abstraction layer is used to abstract the differences between the hardware and the operating systems used for running DPDK. This layer is called the Environment Abstraction Layer (EAL). Through the use of the EAL, applications in DPDK can gain access to low-level implementations for the hardware it is currently running on, by using the generic interface provided by EAL. Additionally, the EAL provides a couple of typical services expected by a developer such as loading and unloading DPDK, System Memory Reservation, utility functions (spinlocks and atomic counters), etc.

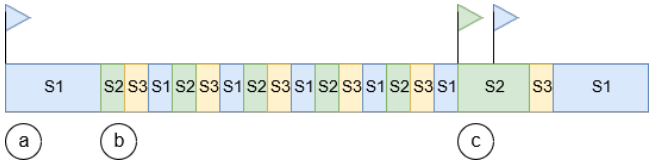

Figure 1. In this example, there are three services running on a single service core. The services are scheduled in a round robin manner, and will therefore always run even if they have no useful work to be done. At a) service S1 has work to be done. At b) the rest of the services will run only to check if there is work to do. At c) both S1 and S2 will have work to do, therefore S1 will have to wait for S2 to complete, and also wait for S3 to run even though it has no useful work to be done.

DPDK applications need to be assigned CPU cores to be able to work. The assigned CPU cores to an application are called *lcores*. Through the use of lcores, the DPDK application will have access to the cores and their CPU cycles. However, since the lcore really is a thread running in user space, the operating system might still interrupt the DPDK application at times.

A packet processing application written in DPDK, will need to access some sort of Poll Mode Driver (PMD). This Poll Mode Driver will allow the application to receive and send network packets.

2.1.2 Poll Mode Driver

One method of implementing a driver for a network interface is by using interrupts. When the Network Interface Controller receives a number of packets, it will send an interrupt to the CPU and force it to take care of the packets. This method is costly because an interrupt almost always requires a full processor pipeline flush. When the packet load is high, an interrupt based system breaks down because the only process that has the chance to execute is the service routine of the interrupt. Hence, there is a limit to the number of packets that can be handled with this approach.

To avoid using interrupts, the New API (NAPI) method was implemented into the Linux kernel. NAPI does not use an interrupt for each packet but instead provides a method to the kernel to fetch all new packets. When the kernel calls this method, it will receive all the packets that have arrived and can do so without the overhead of multiple interrupts[5].

In general, a driver that aims for the highest throughput and the lowest possible latency will need to use polling. With polling, the buffers containing new packets are regularly checked for new packets. In DPDK, the PMDs run in user space but has direct access to the read (RX) and write (TX) descriptors of a Network Interface Controller port. This means that packets can be read from a NIC without interrupting other processes running. However, this also means that packet processing applications will always use up to a hundred percent of the CPU, even when not doing any useful work. As seen as in figure 1.

2.1.3 Packet Processing Applications

A normal packet processing application can run in either of two modes, *run to completion mode* or *pipeline mode*.

In the *run to completion mode*, the application polls the input data structure for packets to process and performs all its operations on the packet and then puts the packet in the send data structure. The packet data structures are designed so that several instances of the application can run on multiple cores simultaneously for increased performance.

In the *pipeline mode*, the processing of the packets is split onto several cores. The first core polls the input data structure and makes part of the processing and the hands the packet over to the next core. The last core in the pipeline puts the packet in the send data structure.

2.1.4 Service Cores

The service core library was created to support the abstraction of specific hardware resources. For example, some systems might have dedicated hardware for encrypting and decrypting packets. But other systems might not and then this functionality must be performed in software using CPU cycles. A developer might then put this task on a service core as a service. This service will then detect and use hardware when available or use a software implementation as a fallback.

Each service runs on a dedicated service core, but there might be multiple services running on a specific service core. These services are scheduled in a simple round-robin run-to-completion. This means all services will get the chance to run and finish their work before the next service will run. Multiple cores can have asynchronous services running. Each service core will loop through its services and execute them. If there are many services running on a core this could potentially lead to high waiting times for some of the services.

This dynamic nature of services means that an app developer might use services as a way to utilize parallelism by adding services dynamically when needed.

However, in the recently released version of DPDK (version 19.05) , the developers would need to manually assign the new services to specific service cores. The developers would need to write specific code for each application and its services. This could lead to services not having enough throughput in a specific time period.

To make sure that this does not happen the common solution is to overprovision the hardware performance. With a load balancer, we propose that the overprovision needed and the amount of developer work might be lessened. Hence, why we are trying to implement a load balancer for service cores.

When running a DPDK application, the developers must specify which of the CPU cores should be used for the application (called lcore mask). By mapping the application worker instances in a 1:1 setup, mapping the first instance to the first lcore and so on, DPDK applications will have full access to the specified cores and will minimize interruptions from

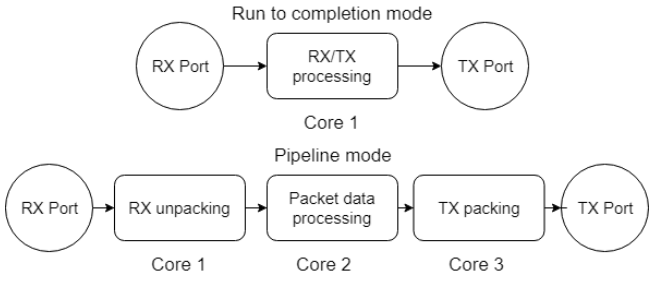

Figure 2. An example of the two packet processing modes.

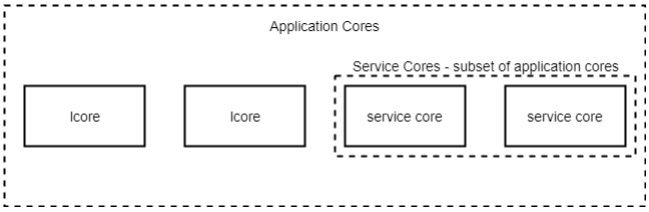

Figure 3. DPDK Service Cores

other processes or the operating system. This makes sure that DPDK applications will always have 100 percent access to the cores specified, and there will be no interruptions from other processes. However, the operating system can still interrupt DPDK applications if needed. Data Plane Development Kit does not use the built-in scheduler in the operating system. It will instead handle its own procedures internally by using a round-robin scheduler.

Service cores, on the other hand, must use a subset of the specified lcore mask to be able to run services. Currently, the developers of the services must manually specify the subset for each service created. It is not possible to use cores outside of the specified ones. The masking process is illustrated in figure 3, showing the number of normal cores and the service core subset.

Because of the way the Service Core library was made, all services are visible to every service core. So there is a need to somehow limit which cores are allowed to run the services. This is done by specifying the cores like previously mentioned. However, internally it will use bit-masks to tag which cores are allowed to run which services. This is updated by the library when the services are specified by the developers at startup.

2.2 Load Balancing the Service cores

Some services, such as packet processing services, will need to keep a reliable throughput in terms of packets sent and received. Therefore, there is a need to find a way to migrate services between service cores to make sure they can always use as much CPU cycles needed without disturbing other services. Because of this, we propose a load balancer specifically made for the DPDK Service Cores library. This load balancer could help developers by making sure they do not have to implement a balancer for each application and service they make.

By using our load balancer, each service core would theoretically run at approximately 100 percent of useful CPU utilization, excluding the current overhead already present for switching between services . There are two general strategies when implementing a load balancer, we will explain and discuss them below.

2.2.1 Static Load Balancing

The first strategy is called "static load balancing". This means that at compile time (or earlier) a schema for assigning services to cores is created. There is a multitude of ways which such a schema could be constructed Currently there is only one method implemented in DPDK.

When developers are creating a service for use in a DPDK system, the developers must specify which cores should act as the service cores. All services will need to be manually assigned, by the developers, to specific service cores.

However, this could be potentially automated. As the DPDK application starts, it could look at all services that are defined, and distribute them across all service cores. The first service would be assigned to the first service core, and as each service gets initialized, they would be assigned to the corresponding service core. If the services exceed the number of available service cores, then the loop will restart and assign the next service to the first service core again - repeating the process for each service. In other words, the load balancing is done at the startup of the DPDK application.

2.2.2 Dynamic Load Balancing

As opposed to Static Load Balancing, it is possible to assign services to specific cores dynamically during run-time. This means that there is a periodic evaluation of which core a service should run on. Ideally, when a service core has finished the loop of running each service, it should check and calculate the workload of itself and compare it to the other service cores. This calculation is done at specific intervals and provides information such as the CPU cycles used by the services during the interval. If a more suitable service core is found, the service will be migrated to it and run on the new Service Core. The services will be migrated when the calculations show that it is more favorable to run on a different core.

One of the reasons why a service core would want to migrate a service is because another service needs more CPU cycles. By migrating a service to a different Service Core, more CPU cycles are available - from the perspective of the measure intervals. One or more services could be fully utilizing the Service Core, therefore a third service would get worse performance because of the waiting time. By moving the third service to a different core, we can be sure the third service will be able to finish faster.

Figure 4 shows an example of running a Dynamic Load Balancer on a multi-core system. A code listing of a pseudo-code implementation of a dynamic load balancer can be seen under section 4.4 on page 6.

Because DPDK services will have a varying CPU utilization, implementing a dynamic load balancer is the best choice. The dynamic load balancer will be able to react in real-time compared to a static load balancer that will not do anything for the increased or decreased CPU utilization.

2.2.3 Thrashing symptom

It is possible for a service to be migrated to a new core, and then ramp up its CPU utilization and trigger the load balancer again. If this happens then the service with the lowest CPU utilization on the core will be moved to a different core. If this service also ramps up its CPU utilization, then the process will repeat and repeat. This process is referred to as "thrashing"[4]. As of right now, there is no way to stop this except waiting for the system to stabilize itself when the CPU load is under the threshold. Therefore, it is of importance that the time between balancing is not too short.

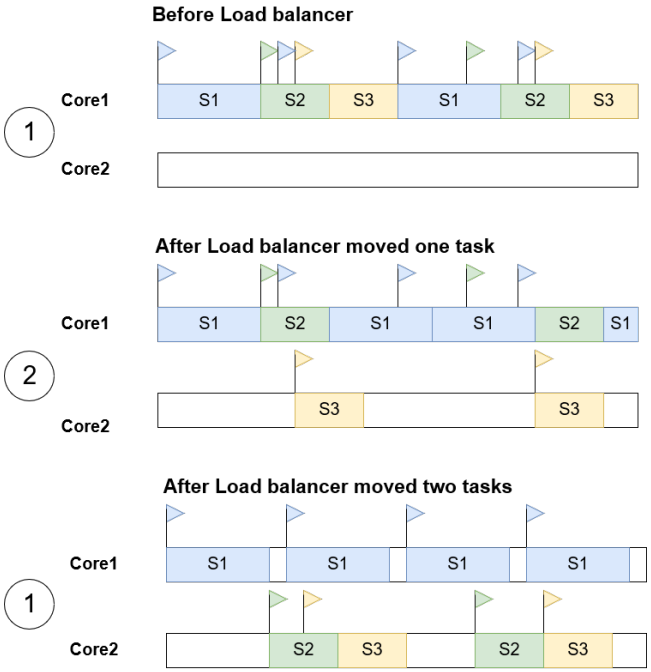

Figure 4. Example of a multi-core dynamic load balancer. In this example, 3 services are running on core 1. The load balancer first moves service 3 to core 2. In the next step, it moves service 2 to core 2. The flags represent when work arrives and ideally work should be finished before the next arrival to avoid filling up buffers. In the last step, this is achieved.

2.2.4 Load Statistics

Using the Dynamic Load Balancing method requires some statistics from the service cores, such as CPU load, amount of services and if the service itself has done some useful work (work that should be used in the calculations). Therefore Load Statistics is a prerequisite for Dynamic Load Balancing, and should also be implemented into the Service Core library.

3. RELATED WORK

The "Tidal Effect" is something worth mentioning as it can impact the network traffic, and thus also increase the load on DPDK services. This effect shows how humans network traffic can be predicted by looking at the time of day. If the predictions are correct, then they can help with scaling up networking functions, such as packet processing [6]. In a patent by Intel [8], the "tidal effect" is also mentioned. The amount of network traffic directly corresponds to the time of day, and what kind of neighborhood the base stations are controlling. However, the PMDs of DPDK makes it hard for the power management modules of the operating system to detect the effect. The patent describes monitoring the receive queue and informing the operating system of a need for more power or possibility to sleep the thread for a while [8]. This could also be used for a load balancer, by preemptively moving services to *correct* service cores before the huge network traffic difference starts to affect the throughput.

Hu et al. [3] present an optimal algorithm for migrating processes so that an equilibrium of load between cores is reached. They compare the balancing problem with that of a heat diffu-

sion process. The heat will spread out until an equilibrium is reached. In the same manner, an optimal load balancer would aim for a per core load that is equal to the average load of all cores and choose the solution that minimizes the need for communication between cores.

Hofmeyer et al. [2] discuss load balancing of processes in contemporary operating systems, mainly Linux. At the time, the load for each core was defined as the number of processes/threads residing in the run queue. The authors define $speed = \frac{executiontime}{wallclocktime}$ as a metric for the load that a process puts on a CPU core. In our work the *speed* metric is used to define the load of a core. However, the load is not measured in seconds but rather in clockcycles. Furthermore, Hofmeyer et al. discuss the interval between balancing and concludes that a benchmark with low cost of migration performed best with an interval of 20ms. However, in the average case, an interval similar to the time quantum of the operating system worked best in practice because of the risk of stale metrics. Because it is not useful to have a universal interval on various systems, in our thesis we had to come up with our own intervals.

4. METHOD

At the start of this thesis, the Service Core library did not have the functionality to gather load statistics for its service cores and services. Therefore the first thing to do was to change the service structure to be able to report back useful work. When this is implemented, the next step would be to implement a statistics gathering functionality that can serve as the base for the load balancer algorithm.

4.1 Changing the service structure

DPDK services are actually functions, i.e callbacks, that runs on service cores specified by the programmer. Because of this the structure already implemented did not have the capabilities of calculating how much each service uses the CPU cycles given to it. To make statistics gathering work, there is a need to change the original structure to make sure each service keeps track of their own used CPU cycles. Therefore a global list is added which each service core can access, this allows for load statistics to be gathered. With the global list, it is possible to calculate the total CPU utilization per Service Core, excluding the overhead.

The Service Core library has multiple data structures that are used to abstract the services and their callbacks. Therefore it makes sense to make each service tell the library if they did anything useful when they finish their callback. This is simply done by either returning true or false from the callback. With the changes to the service return values, it is possible to know if a service has done useful work or not. If a service callback returns false, then the used CPU cycles will be dismissed.

4.2 Gather load statistics

Because the load balancing implementation will need to measure load between time periods, a CPU cycle calculator is implemented. The calculator works by taking a CPU cycle timestamp before running the service callback, and again after the callback has returned. The difference between these two timestamps will give us the CPU cycles used by the service. The difference will be stored in the service data structure for later use.

```
struct rte_service_spec_impl {
 /** Removed unrelated code ** /

 /* per service statistics */
 rte_atomic32_t num_mapped_cores;
 uint64_t calls;
 uint64_t cycles_spent;
 uint8_t active_on_lcore[RTE_MAX_LCORE];

 /* load balancer specific classifiers*/
 uint64_t work_cycles_spent;
} __rte_cache_aligned;
```

Because the load balancer will do its work between time periods, a simple frequency clock check is needed. By using the CPU's clock frequency, it is possible to approximately know when a certain time has passed. Time periods are hard to set for a universal system like a load balancer. Each system will be different from our test system. Therefore, we have chosen to use a time period of 250 microseconds. In future implementations, this should be changed for the system it is currently running on. After each time period, the used CPU cycles for each service will be saved. Every 5 milliseconds, the load balancer algorithm will be triggered, and in turn also reset all of the statistics - restarting the process. A limit at 5 milliseconds is chosen because, in our test environment, this will allow the load balancer to have a high amount of data to use for its calculations.

4.3 Creating the test application

Before a load balancer can be implemented a test-application must be created. Our aim was to have a test application that could use up to 5 registered services running on up to 5 cores. With this application, we would be able to test how a load balancer would move services between cores to achieve the desired performance goal. One of the problems with load simulations is that the workload is not a single chunk of work but rather a stream of packets that arrive in a somewhat random pattern. To simulate this kind of workload, the *Leaky bucket* method was used [7].

4.3.1 Token-based test application

The *leaky bucket* method creates tokens for each service to use. After reaching an upper limit the services can withdraw their corresponding tokens and simulate work proportionally to the number of tokens withdrawn. The simulation of the work was done by looking at the CPU Time Stamp Counter as many times as tokens were withdrawn. With this method, the services could be limited to a certain percentage of CPU load by only filling the buckets at a rate corresponding to the desired load.

The first test application was written using the leaky bucket method. In this application, three services and two processor cores were used. The pattern for the bucket fill rate was chosen so that we make sure that the core with two services will periodically reach the threshold of 70 % and hence trigger a balancing event.

Measuring latency between when a job should start and when it actually is starting/finishing is not sufficient to determine if

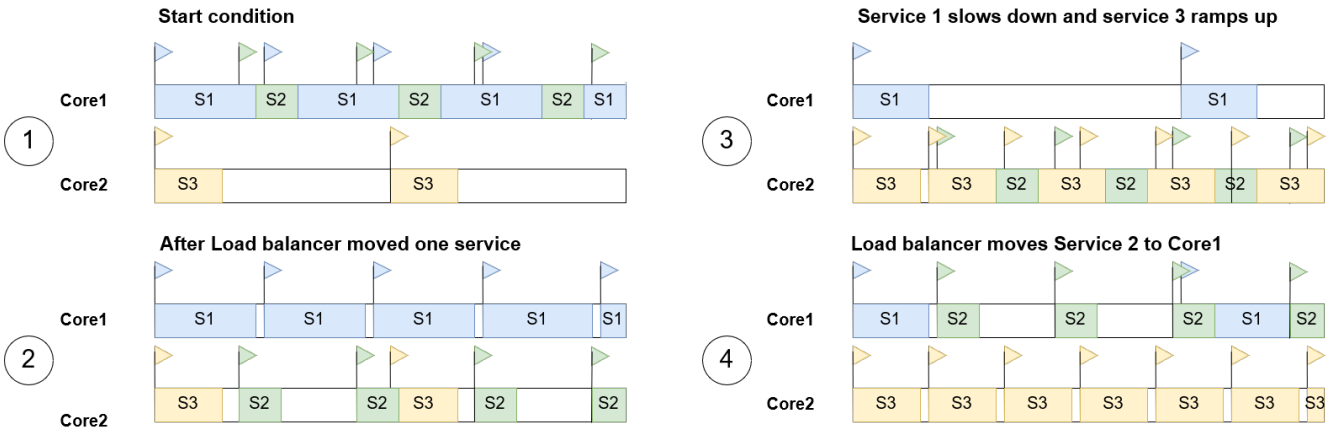

Figure 5. Example of the implemented load balancer. Service 1 and 2 reside on Core 1 at start up, making the services miss their deadlines. By moving Service 2 to Core 2, all of the deadlines can be held. However, as Service 3 gets higher load, it will make Service 2 miss its deadlines again. As Service 1 has lower load now, the load balancer will move Service 2 back to Core 1.

the load balancer gives an improvement to the performance. And, it has some drawbacks when running the same service on several cores, e.g. it has to be multi-thread safe. Therefore, a second test application needed to be implemented.

4.3.2 Deadline-based test application

This application is based on deadlines. Each service has a deadline for when the next packet of work will arrive. The service will then calculate the next deadline based on the amount of work the service is supposed to do and the predefined load that we want the service to aim for. To make the deadlines behave a bit more like real-world network traffic, random jitter is added to the deadlines. And, in order to simulate the tidal effects of network traffic, it is possible to dynamically alter the load used for calculating the next deadline. By dynamically altering the load, we can make the traffic ramp up and/or down to trigger the load balancer when reaching the upper threshold of CPU utilization.

The deadline based test application also has an advantage when it comes to measuring performance. It is easy to measure how much a service overshoots its deadline, and predict packet-drop and latency using the measurements.

4.4 Implementing the load balancer

Finally, after implementing the statistic gathering functionality and the test application, the load balancer is now possible to implement. At first, an upper CPU utilization threshold is needed. This threshold will trigger the load balancer to do its calculations and migrations when reached. This threshold will be looking at the statistics gathered across all time periods, and calculate the average CPU utilization for all service cores.

When choosing the threshold some factors need to be considered. The threshold cannot be too low so that services move too often, and it cannot be too high so that services are never moved. As a compromise, we have chosen a 70% core load as the threshold for our load balancer.

Because the Service Core library runs on each specified core, the load balancer will also run on each specified core. Which means that the load balancer will become a sort of a distributed load balancer.

Pseudo-code for a dynamic load balancer.

```
if(time_to_balance_load) {
  core_load = useful_cycles_spent / total_cycles;
  if(core_load > threshold) {
    service_to_move = find_lowest_load_service();
    if(service_to_move == NULL) continue;
    receiving_core = find_lowest_load_core();
    if(receiving_core == NULL) continue;
    if(new_core_load < threshold)
      move_service;
  }
}
```

When the upper threshold is reached, the load balancer checks the CPU utilization of the other service cores. To be able to migrate a service, the service must not be alone on a core - as this will trigger the thrashing[4] symptom of moving services all the time. If the service is multi-threaded, it is not allowed to migrate to a core with an already running instance of the same service. The load balancer selects the core with the lowest CPU utilization and only migrates the service to it if the CPU utilization afterward will not trigger the upper threshold again. This is to make sure the load balancer will not be triggered again and waste CPU cycles by doing unnecessary calculations. The migration happens by altering the existing service mask. The service mask for the new core will be changed to include the service, while the service mask for the old core will be changed to remove the service. This way a seamless migration can be made, the only overhead is the operations needed to change the service masks.

By having the load balancer be run after the service loop is done, there is no need to check if the service is currently being run. Even if the service is multi-threaded, the load balancer will still not check if the service is being run. As the load balancer is moving the service from an "idle" service core (a service core currently running the load balancing code), it can always be sure the service is not running on the core. And because multi-threaded services cannot live on the same core,

the load balancer can be sure the new core will not have an instance of the service running.

When migrating services, the service with the lowest CPU utilization will be migrated. The reasoning to this is a low CPU utilization will make sure the other high CPU utilization services are able to keep up with the work. By migrating the service with the least CPU utilization, the impact on performance (while migrating) shouldn't be huge as the service isn't run that often either way. This migration procedure can be seen in figure 5.

4.5 Calculating the differences

Lastly, after implementing all the needed functionality, it is now possible to plot the results. Because each service in our test application has a deadline (when it should start simulating work), it is possible to calculate the difference between the deadline and when the service actually started to simulate work. This difference gives us a way to measure if the load balancer helps with the latency between each service run. These differences are saved to a simple CSV file, which can then be used to plot graphs. In our case, we used the programming language R.[3]

By using the data in the CSV file, it is possible to calculate the mean difference between applications that use the load balancer and applications that do not.

5. RESULTS

5.1 Token-based test application

Results from the first, token-based, test application indicate that the load balancer works as intended. This test was specifically designed to trigger the load balancer to move services between cores. Logs from the application show that the services were migrated as intended..

5.2 Deadline-based test application

Results from the second deadline-based test application showed both decreased latency and an increased number of processed packets.

5.2.1 Decreased latency

By running two services on the same service core, and then increasing the load in 15% intervals, it is possible to see the difference in the latency between services. In figure 6, the y-axis shows the number of CPU cycles a service is delayed and the x-axis shows the load corresponding to the delay. The lower the delay, the better it is for the services. If the delay is high, it would mean that a service has to wait much longer to be able to run and complete the task given to it.

Figure 6 shows that the load balancer improves the latency by making it almost 10 times smaller than without the load balancer. By having a load balancer implemented and used, it will make the services perform better by migrating them when needed.

Because the two services are running on the same core from start-up, they will trigger the load balancer at a 35% load per service. This will make the load balanced test application handle more load because each core will be running at 35% instead of 70%, after migration. This also means each service will have more CPU cycles available to them than previously. As seen in figure 6, the load balanced test application can handle more deadlines before the delay starts to rise due to using all available CPU cycles. The non-load balanced test application will instead start to get huge delays at a 35% load per service. This is because the services will still be running on one core only, and share the available CPU cycles between them. As these CPU cycles are much lower than the load balanced test application, it is safe to say that the load balancer gives a huge improvement latency-wise for DPDK in this scenario.

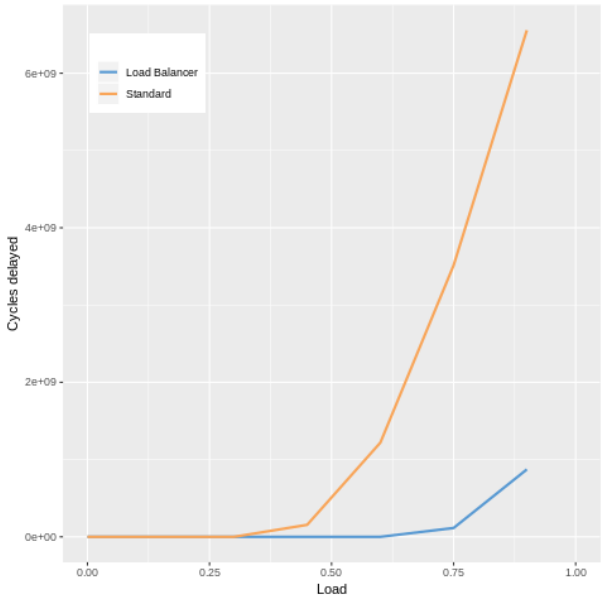

Figure 6. The total CPU cycle delay for two services between deadlines. The load pictured is per service, therefore a load on 50% would mean a total load on 100% if they are running on the same core. Lower delay is better.

5.2.2 Increased packet processing

Another result from the deadline-based test application was an increase in packet processing. In one test we ran 2 services for 2 seconds with a load that ramped up over time. The test was run first with the load balancer enabled and then disabled (standard). With the load balancer enabled, one of the services was migrated to another core and neither service had to wait for the other to finish. This lead to a higher total amount of packets processed. When the load balancer was not enabled, the services took turns in processing packets. Figure 7 show the amount of packets processed in each scenario. In this particular test, when the load balancer was enabled almost 2 million more packets were processed than with the standard implementation. This represents an increase of almost 40% in this particular test.

5.3 Multi-service scenario

In addition to the simple deadline-based test application we also recreated the load scenario of first leaky bucket test appli-

[3]https://www.r-project.org/

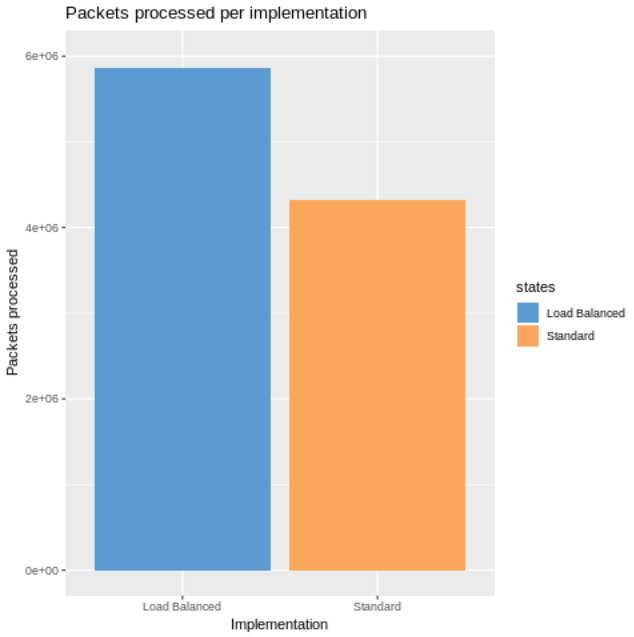

Figure 7. The difference in packets processed between the standard implementation and the load balancer implementation. The test was conducted with two services running with two cores available and during 2 seconds with dynamically increasing load.

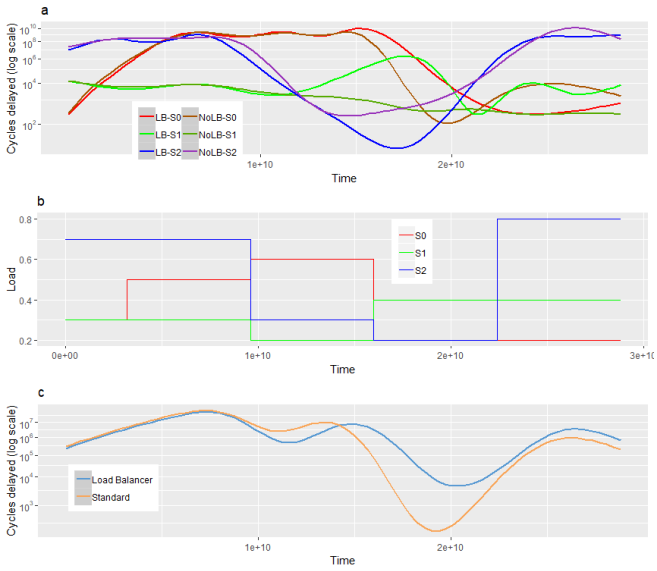

Figure 8. The top plot (a) shows delay with 3 services running on two cores(Note that the y-axis has a log scale.). Both the delays when using the load balancer and without the load balancer is shown. The dynamic load applied is shown in the middle plot (b). The bottom plot (c) shows total delay with or without the load balancer. When the test starts services S0 and S1 are running on the same core and S2 on its own core. At $1 * 10^{10}$ service can S1 move to the other core.

cation but with the deadline based approach. Figure 8 shows metrics from this scenario. The top plot shows the amount of cycles a service is delayed, with the load balancer and without it. In the middle plot, the load of the services are depicted. As the load is the same with and without the load balancer, we can save space and only draw one line for each service. We can also see how the total cycles delayed (all of the services delayed cycles) are affected when the load balancer is running, and when it is not running. Here we can see an improvement when using the load balancer, as in the total cycles delayed drops when all services are balanced.

We can also see that the load balancer will not impact the total cycles delayed unless the load is above the threshold. However, by looking at figure 8, we can see that there could be room for improvement. More about this later in section 6.4.

6. DISCUSSION

6.1 Results

6.1.1 Token-based test application

The purpose of the first test application was to show if the load balancer would have the correct behavior. However, it also indicated an increase in performance. When the core load reaches 100% then the buckets accumulates tokens and when the fill rate decreases it takes some amount of time before the buckets return to normal. The time it takes to return to normal (e.g the time it takes for the core load to return to the level of the bucket fill rate) can be roughly measured from the data and indicates that the balancer is indeed working as intended and is providing better performance.

6.1.2 Token-based test application

The second test application was constructed to be able to measure the the time between when a job arrives and when a service started processing it. In a scenario with two services on one core it is not surprising that the the delay increases rapidly when the load for each service get close to 50% since this means that the total load of the core would be 100%. In a scenario with equal amount of cores and services (the blue line in figure 6) the delay is low until the the core load reaches 75%. This indicates that choosing a threshold of 70% is a fair choice.

6.2 Service masks

The service mask is used as a way to keep track of which core a service *may* run on. However, since the applications do not really have any restrictions of this kind it was decided that the service mask would be reused as a means to track which service is currently running on a core. This also allowed for seamless migration of a service to a diffrent core. This significantly reduced the time to implement the load balancer as a proof of concept. However, in a production environment, there is a need to both keep track of where a service is running and where it *may* run, so an extra mask would be needed to be able to track this.

6.3 Consequences with moving services

The choices made for the implementation of the load balancer has some consequences that needs to be addressed.

6.3.1 Lowest CPU utilization

In the load balancer the service with the least impact on the core load was moved. Hypothetically, if three services are running, with a load of 60%, 15%, and 5% respectively, then

the load balancer will be triggered two times. Once to move the service with the 5%loads, then one more time to move the remaining 15% load service. A more advanced load balancer might migrate several services at once but this will come with the cost of more calculations to find a good fit.

6.3.2 Ethical dilemmas regarding power consumption

Since applications of the type that we investigate in this thesis are supposed to be used in a high number of units it is important that energy is not wasted. In this thesis, we focus on performance and how to migrate services between cores to achieve lowest average latency (and throughput) with the number CPU cores available. However, as will be discussed in section 8.3 *scale-up and scale-down*, it is also of importance to be able to scale down services to run on fewer cores when the CPU utilization is low. This will, in turn, make it possible to put cores into sleep mode and save energy.

6.4 Using the upper threshold

By looking at figure 8, we can see that the load balancer only affects the delay significantly around cycle count $1.5 * 10^{10}$. This is because the upper threshold, of the load balancer, is reached at this cycle count. Therefore, the services are balanced and can work more effectively. However, there is room for improvement as mentioned in section 5.3. Specifically by altering the load balancer to look at all of the services' delays, and then balancing them according to this instead of core load. This is something that could be done for a future implementation, and is talked about in section 8.1.

7. THREATS TO VALIDITY

Some aspects of the validity of this work need to be addressed. Most of the threats are associated with the simulated workloads. Workloads will most likely be different from one application to another and to obtain results with better accuracy one would have to model workloads corresponding to the target application. However, the following threats to validity must be kept in mind when reading the results.

7.1 Dropping packets in real-life applications

When the core approaches a load of 100% it gets more and more likely that the next deadline has already been passed. This means that packets are entering faster than they can be processed and are therefore filling up buffer queues. This means that the waiting time is correlated with how many packets are residing in the buffer. In a real-life situation, there would be a finite limit to how many packets that fit in the buffer and when reaching this limit packets would start to be dropped. However, the size of these buffers would be dependent on the application and target hardware and hence not something we can account for in our test applications.

7.2 Packet processing performance

Because our test applications are running in a ideal situation, the performance increase of packet processing, mentioned in section 5.2.2, could be lower in real-life. In real-life there are multiple variables that affects the packet processing, which are not present in our test environment. For instance, the test applications are only simulating work and only affect the content of the caches slightly. This means that the load balancer could affect the caches significantly more in a real production environment. This should therefore be kept in mind when looking at the results.

7.3 Non-Uniform Memory Access Cores

In a real-world situation, it might not be possible to move a service if it depends on hardware that is only available on a specific core. This could, for example, be allocated memory, directly connected to a specific core. Migrating such a service to another NUMA core would bring a heavy performance hit if it would still need to use memory attached to another NUMA core. These restrictions should be addressed by the use of the service mask and using other data structures to keep track of which core a service runs on.

8. FUTURE WORK

Some advanced features were not implemented into this version of the basic load balancer. But for future implementations, these features should be investigated and implemented. These features are specific to the load balancer created in this thesis. Therefore our own suggestions on how to implement these features are also talked about.

8.1 Using the delay as metric

In this thesis, we have mostly considered processor load as a metric for when to balance the services. As mentioned in section 6.4, an alternative to using the CPU utilization as a threshold could be using the delays as the threshold. By calculating the delays and balancing the services when the delay is too high, it could be possible to make the load balancer step in earlier and improve the throughput even more. This is something that should be looked upon in a future implementation.

8.2 Frequency scaling

The fastest way to change the performance of a CPU is by altering the running clock frequency. The round robin, polling nature of the services running on service cores does not allow for the operating system to tweak the frequency of a CPU core. However, the load balancer has all the metrics to be able to manage this. At times with low load the load balancer could set a lower frequency to save power and when the load increases it could raise the frequency accordingly.

8.3 Scale-up and Scale-down

If a service is written so that multiple instances of it can run at the same time i.e. multi-thread safe, it is possible to increase performance not only by increasing frequency or moving a service but also by launching more instances of this service on other cores. The load balancer should also be able to automatically scale down the cores used by the services when the CPU utilization is lower than some scale-down threshold. I.e. When a certain time has passed, and the cores are not being utilized more than, for example, 20%, then the load balancer should migrate services to fewer cores so that some cores might be put to sleep. If multiple instances of a service are running on several cores then one or more could be halted. A load balancer that scales down in this way will use less power. This lower power draw allows for an increase in processor frequency. Applications that need a high serialized

performance will benefit from this increase in frequency. This is something that could be investigated further.

8.4 Testing in a production environment

Because our load balancer is used in ideal test environments, there is a need to test it in a live production environment. By testing the load balancer in a production environment and comparing the real-life results with the results found in this thesis, a better conclusion can be given about using a load balancer for different environments. Some production environments may benefit more from a load balancer than other environments. Therefore, we recommend testing the load balancer in various situations and environments.

8.5 Service mask usage

The service mask was not intended to be used in the way that the currently implemented load balancer does. It should instead use the service mask as intended i.e. keeping track of which cores the services are allowed to run on. This requires another data structure that keeps track of which core a service is running This will help by always remembering what the developers specified at start-up (e.g. to avoid migrating out of NUMA-clusters), and make it easier to migrate services later on.

9. CONCLUSION

A basic dynamic load balancer was implemented into Data Plane Development Kit. When testing different scenarios in our test environment, we can see a performance increase in packet processing and lower latency for each service. However, our test environment is in an ideal state, meaning it does not have any of the real-life variables that could potentially negate the performance improvements. Therefore, we recommend testing each production environment with the load balancer to determine in which scenarios it would be useful. The results from testing the production environments could show that specific environments benefit more from a load balancer than other environments.

As mentioned before, the results from our test environments show that the introduction of a load balancer could potentially help network operators to handle increased load more effectively. Even the performance hits from natural effects, such as "tidal effect", could be negated. The performance increase would mean that the end users would get lower latency in network traffic. It could also mean network operators could potentially save money by using the same hardware as much as possible before adding more processing power.

However, future implementations may want to look at using the service delay as a metric instead of using an CPU utilization threshold. As discussed in section 6.4 and section 8.1, this could be an improvement to the load balancer. By switching the metric used, the load balancer can instead step in and balance the services when the delay is higher than wanted. This would mean that the load balancer instead focuses on delay instead of CPU load.

Overall, the work in this thesis showed a need for a load balancer, that can bring performance improvements to Data Plane Development Kit Service Cores.

10. REFERENCES

1. Ericsson. 2018. Ericsson Mobility Report November 2018 – Ericsson. (Nov. 2018). https://www.ericsson.com/en/mobility-report/reports/november-2018

2. Steven Hofmeyr, Costin Iancu, and Filip Blagojević. 2010. Load Balancing on Speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 147–158. DOI: http://dx.doi.org/10.1145/1693453.1693475 event-place: Bangalore, India.

3. Y. F. Hu, R. J. Blake, and D. R. Emerson. 1998. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience* 10, 6 (1998), 467–483. DOI: http://dx.doi.org/10.1002/(SICI)1096-9128(199805)10:6<467::AID-CPE325>3.0.CO;2-A

4. Peter J. Denning. 1968. Thrashing: Its Causes and Prevention. *AFIPS Conf. Proc.* 33 (01 1968), 915–922. DOI:http://dx.doi.org/10.1145/1476589.1476705

5. Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 2001. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5 (ALS '01)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1268488.1268506 event-place: Oakland, California.

6. Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. 2010. Limits of Predictability in Human Mobility. *Science* 327, 5968 (Feb. 2010), 1018–1021. DOI: http://dx.doi.org/10.1126/science.1177170

7. J. Turner. 1986. New directions in communications (or which way to the information age?). *IEEE Communications Magazine* 24, 10 (Oct. 1986), 8–15. DOI:http://dx.doi.org/10.1109/MCOM.1986.1092946

8. Danny Y. Zhou, Mark D. Gray, and John J. Browne. 2017. Techniques for power management associated with processing received packets at a network device. (Aug. 2017). https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20170801&DB=&locale=en_EP&CC=CN&NR=107005531A&KC=A&ND=4 CIB: H04L29/06.