Metronome: adaptive and precise intermittent packet retrieval in DPDK

Marco Faltelli
University of Rome Tor Vergata
marco.faltelli@uniroma2.it

Giacomo Belocchi
University of Rome Tor Vergata
giacomo.belocchi@uniroma2.it

Francesco Quaglia
University of Rome Tor Vergata/CNIT
francesco.quaglia@uniroma2.it

Salvatore Pontarelli
Axbryd/CNIT
salvatore@axbryd.com

Giuseppe Bianchi
University of Rome Tor Vergata/CNIT
giuseppe.bianchi@uniroma2.it

ABSTRACT

DPDK (Data Plane Development Kit) is arguably today's most employed framework for software packet processing. Its impressive performance however comes at the cost of precious CPU resources, dedicated to continuously poll the NICs. To face this issue, this paper presents Metronome, an approach devised to replace the continuous DPDK polling with a sleep&wake intermittent mode. Metronome revolves around two main innovations. First, we design a microseconds time-scale sleep function, named `hr_sleep()`, which outperforms Linux' `nanosleep()` of more than one order of magnitude in terms of precision when running threads with common time-sharing priorities. Then, we design, model, and assess an efficient multi-thread operation which guarantees service continuity and improved robustness against preemptive thread executions, like in common CPU-sharing scenarios, meanwhile providing controlled latency and high polling efficiency by dynamically adapting to the measured traffic load.

ACM Reference Format:
Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. 2020. Metronome: adaptive and precise intermittent packet retrieval in DPDK. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3386367.3432730

1 INTRODUCTION

Packet processing is a very common task in every modern computer network, and Data Centers allocate relevant amounts of resources to accomplish it. Also, DPDK is the most used framework for software packet processing, as it provides excellent performance levels [1]. On the downside, deploying DPDK applications comes with a series of shortcomings, the major one being the need for fully reserving a subset of the available CPU-cores for continuously polling the NICs—a choice that has been made in order to timely process incoming packets. This approach not only gives rise to constant, 100% utilization of the reserved CPU-cores, but also leads to high power consumption, regardless of the actual volume of packets to be processed [2].

Indeed, there are many reasons which suggest that the availability of solutions capable to replace continuous polling with an intermittent, sleep&wake, CPU-friendly approach would be beneficial. While Google states that even small improvements in resources utilization can save millions of dollars [3], previous work has brought about evidence that despite Data Center networks are designed to handle peak loads, they are largely underutilized. Microsoft reveals that 46-99% of their rack pairs exchange no traffic at all [4]; at Facebook the utilization of the 5% busiest links ranges from 23% to 46% [5], and [6] shows that the percentage of utilization of core network links (by far the most stressed ones) never exceeds 25%. Dedicating a full CPU, the most greedy component in terms of power consumption [7], to continuous NIC polling thus appears to be a significant waste of precious resources that could be exploited by other tasks. To a greater extent this appears the case nowadays: CPU performance is struggling to improve and seems about to reach a stagnation point [8, 9], at the moment of time in which CPUs burden is ever growing, also because of newly emerging needs for security (e.g. the Kernel Page Table Isolation—KPTI–facility adopted by Linux to prevent attacks based on hardware level speculation, like Meltdown [10]).

DPDK's continuous CPU usage may also raise concerns in multi-tenant cloud-based deployments, where customers rent virtual CPUs which are then mapped onto physical CPUs in a time-sharing fashion. In fact, fully reserving CPUs for DPDK tasks complicates (or makes unfeasible) the adoption of resource sharing between different cloud customers. Also, 100% usage of computing units is not favorable to performance in scenarios where threads run on hyper-threaded machines—just because of conflicting usage of CPU internal circuitry by the hyper-threads. Hence, multi-threading should be avoided in continuous polling-based DPDK deploys, posing the additional problem of making this framework not fully prone to scale on off-the-shelf parallel machines. While major cloud providers [11, 12] have already enabled the deployment of DPDK applications in their data centers, to the best of our knowledge such solutions still present the shortcomings of drivers based on continuous-poll operations.

To face these issues, this paper proposes Metronome, an approach devised to replace the continuous DPDK polling with a sleep&wake intermittent function. Albeit this might seem in principle an obvious idea, in practice its feasibility was so far hindered

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12.
https://doi.org/10.1145/3386367.3432730

by the lack of a precise sleep function in the microseconds time-scale, a gap which we overcome in this paper with the design of a new microseconds time-scale sleep function, named hr_sleep(), which significantly outperforms the Linux nanosleep() of more than one order of magnitude in terms of precision. Metronome revolves around a novel architecture and operating mode for DPDK where incoming traffic, even if coming from a single receive queue, is shared between multiple threads which dynamically switch from polling the receiving queue to sleeping phases for short and tunable periods of time when the queue is idle. Owing to a suitable adaptation strategy which tunes the sleeping times depending on the load conditions, Metronome achieves a stable tunable latency and no substantial packet loss difference compared to standard DPDK while reaching significant reduction for both CPU usage and power consumption. More specifically, our contributions summarize as follows:

- we engineer hr_sleep(), an optimized implementation of a high resolution timers based sleep service in Linux, obtaining 15x gain in precision for μs grained sleep periods of common-priority time shared threads, with respect to the standard implementation;
- we experimentally show that the increased precision of hr_sleep(), opposed to the limited precision of prior state of the art, enables a precise sleep&wake operation as a viable alternative to the expensive NIC's continuous polling;
- leveraging our high resolution and precise sleep service, and a few additional facilities in the ISA—specifically the ones supported by x86 CPUs—we enable Metronome, a novel multi-threaded architecture for DPDK applications, which offers low overhead thread-coordination and dynamic CPU allocation to DPDK threads, obtaining lower CPU usage compared to standard DPDK settings, as well as better capability to exploit hardware level parallelism. As an indirect effect, Metronome also has the capability of positively impacting energy efficiency under specific workloads;
- we present an analytical model for Metronome, which is used for driving the CPU resource allocation to threads within the polling scheme, making the DPDK framework dynamically adapt its behavior (and its demand for resources) to the workload;
- we extensively assess Metronome on 10 gigabit/s NICs, in various load conditions, and we test its integration in three different applications: L3 forwarding, IPsec, and FloWatcher [13], a high-speed software traffic monitor.

2 RELATED WORK

When processing the packet flow incoming from NICs, two orthogonal approaches can be exploited: (continuous) polling and interrupt. Polling-based frameworks can either rely on a kernel driver (e.g. netmap [14], PFQ [15] and PF_RING ZC [16]) or bypass the kernel through a user space driver, like DPDK [17] and Snabb [18]. Such frameworks rely on high performance, batch transferring mechanisms such as DMA and zero copy [16], preallocating memory through OS hugepages. Among all of these solutions, DPDK has definitely emerged as the most used one, as it reaches the best performance levels [1]. Furthermore, it is continuously maintained by the Linux Foundation and other main contributors (e.g., Intel).

As mentioned, one of the main shortcomings of DPDK is the excessive usage of resources (CPU cycles and energy), caused by the busy-wait approach used by threads to check the state of NICs and Rx queues. Intel tried in [19] to address the energy consumption issue via a gradual decrease of the CPU clock frequency under low traffic for a commonly used application such as the layer-3 forwarder. A similar approach is used in [20], with the addition of an analytical model exploited to choose the appropriate CPU frequency. Along this line, [7] proposes a power proportional software-router.

However, while the clock frequency downgrading approach allows achieving minimal power consumption [7] without noticeably affecting performance, these solutions do not take into account another crucial aspect, namely the actual usage of CPU. In fact, downgrading the clock frequency of a CPU-core that is anyhow dedicated to a thread operating in busy-wait (namely, continuous polling) mode still implies 100% utilization. Hence, the CPU-core is anyhow unusable for other tasks. Moreover, downgrading the clock frequency of CPUs is not feasible in cloud environments since (i) they are shared between different processes and customers and (ii) providers would like them to be fully utilized in order to reach peak capacity on their servers [21]. Our proposal bypasses these limitations since we do not rely on any explicit manipulation of the frequency and/or power state of the CPUs. Rather, we exploit an optimized operating system service, to control at fine grain the timeline of CPU (and energy) usage by DPDK threads—hence the name Metronome—which are no longer required to operate in busy-wait style. Such control is also based on an analytical model, that allows taking runtime decisions on the basis of the packet workload variations.

At the opposite side, the literature offers interrupt-based solutions. However, the huge improvements of NICs (1GbE to 100 GbE), and the contextual stall of CPU performance because of the end of Moore's Law and Dennard Scaling [8, 9] has evidenced performance limitations of the interrupt-based approach. In fact, interrupt-based solutions suffer from the latency brought by the system calls used to interact with the kernel level driver managing the interrupts, packet copies to user space and so on. Moreover, an interrupt-based architecture operating at extreme interrupt arrival speed may cause livelocks [22]. The Linux NAPI aims at tackling these limitations by providing an hybrid approach which tends to eliminate receive livelocks by dynamically switching between polling and interrupt-based packet processing, depending on the current traffic load. Such a mechanism currently works only for kernel-based solutions, not for user space ones, like DPDK. XDP [23] is a framework built inside the Linux kernel for programmable packet processing. Instead of moving control out of the kernel (with the associated costs), XDP acts before the kernel networking stack takes control so as to achieve latency reduction. While XDP provides some significant benefits such as total integration with the OS kernel, improved security and CPU usage proportional to the actual network load, it still does not quite match DPDK's performances [23] and currently supports less drivers than DPDK does [24, 25]. Our solution is instead fully integrated with DPDK. Works like Shenango [26] and ZygOS [27] explicitly target latency sensitive applications, while other contributions try to accelerate packet processing by moving computation to modern NICs [28–30]. Finally, the benefits coming from our hr_sleep() could be also employed in solutions regarding traffic shaping policies [31–35].

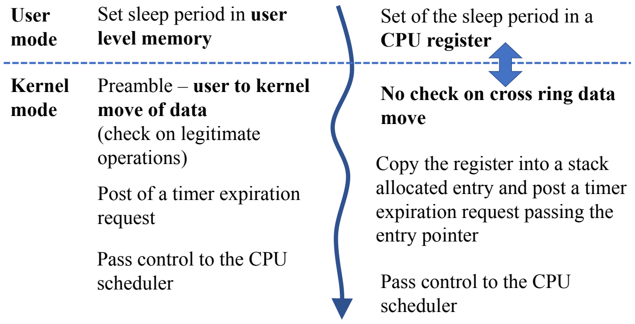
3 METRONOME ARCHITECTURE

3.1 Fine-Grain Thread Sleep Service

One of the key points in our solution is the inclusion of an innovative implementation of a fine-grain sleep service in Linux. This type of service, and the actual precision of the supported sleep interval, is essential for the construction of any solution where the following two objectives need to be jointly pursued: 1) threads must leave the CPU if there is currently nothing to do (in our case by the side of packet processing); 2) threads must be allowed be CPU rescheduled—gaining again control of the CPU—according to a tightly controlled timeline. Point 2) would allow the definition of an architectural support where we can be confident that threads will be able to be CPU dispatched exactly at (or very close to) the point in time where we would like to re-execute a poll operation on the state of a NIC—to determine whether incoming packets need to be processed. On the other hand, point 1) represents the basis for the construction of a DPDK architecture not based on full pre-reserving of CPUs to process incoming packets.

In current conventional implementations of the Linux kernel, the support for (fine-grain) sleep periods of threads is based on the nanosleep() system call, which has index 35 in the current specification of the x86-64/Linux system call table. The actual execution path of this system call, particularly at kernel side, is shown in Figure 1a. When entering kernel mode the thread exploits two main kernel level subsystems. One is the scheduling subsystem, which allows managing the run-queue of threads that can be rescheduled in CPU. The other one is the high-resolution timers subsystem, which allows posting timer-expiration requests to the Linux kernel timer wheel. The latter is a data structure that keeps the ordered set of timer expiration requests, so that each time one of these timers expires the subsequent timer expiration request is activated. The expiration of a timer is associated with the interrupt coming from the High Precision Event Timer (HPET) on board of x86 processors. Hence, when one of these interrupts arrives, the handler reactivates HPET for a subsequent interrupt request based on the residual time of the next timer expiration. For the case of sleep services, the timer expiration leads to the processing of a callback function that brings the sleeping thread back onto the run queue, thus making it CPU-reschedulable.

nanosleep()'s limitations: what we have noted in this architecture is that the very early part—the preamble—of the execution of the kernel level code that implements the nanosleep() system call is not fully prone to support a "precise" fine-grain sleep phase of threads. More in detail, the nanosleep() system call requires specifying the duration of the sleep phase through a data structure (a table), called struct timespec, that is kept in user space and is populated by the calling thread (the one that needs to sleep) before the system call is called. Therefore, this thread passes to the nanosleep() system call the pointer to the populated data structure. However, the actual content of the data structure needs to be then read by the kernel side software that performs a user-to-kernel data move operation.


User mode
Set sleep period in **user level memory**
Kernel mode
Preamble – **user to kernel move of data** (check on legitimate operations)
Post of a timer expiration request
Pass control to the CPU scheduler
Set of the sleep period in a **CPU register**
No check on cross ring data move
Copy the register into a stack allocated entry and post a timer expiration request passing the entry pointer
Pass control to the CPU scheduler

(a) nanosleep (b) hr_sleep

Figure 1: Sleep services: thread execution steps.

The latter operation is well known to be problematic in operating system architectures, since it might lead to break the ring model, and kernel isolation from user activities. Hence a kernel-level infrastructure is used in Linux to determine whether the data move operation is legitimate. This requires executing several services within the kernel that lead to: 1) additional machine instructions to be executed right before starting the actual thread sleep and 2) the possibility for the calling thread to be preempted while running this preamble. As for point 2), we could be in presence of a thread that would like to sleep for a very fine grain period (thus willing to almost immediately leave the CPU spontaneously) but we bring it off the CPU before the actual request can be finalized (thus delaying the corresponding insertion of the timer expiration request into the timer wheel). In other words, the thread would need to be CPU-rescheduled again before being able to finalize its sleep request. This would lead to: (A) unpredictability/stretch of the actual length of the overall interval between the invocation of the sleep service and the actual wakeup, and to (B) the usage of more CPU-cycles (and corresponding energy consumption) just for telling the kernel that the thread would simply like to leave the CPU for a while. Clearly, increasing the priority of the thread can partially alleviate this problem—since the likelihood of preemption can be reduced. However, this approach would reduce the degree of freedom in the configuration of CPU scheduling policies, hence revealing not an adequate solution in contexts where the infrastructure owner wants to actuate CPU sharing across disparate applications (like when hosting multiple VMs on top of hypervisors). In any case, independently of whether preemption will occur, the CPU cycles spent for that preamble lead to a delay for the post of the timer-expiration request to the timer wheel, leading the thread to actually start its timed-sleep phase with a delay.

One motivation for this implementation of nanosleep() is that the struct timespec data structure allows specifying intervals using different fields that correspond to different units (such as seconds or nanoseconds).

hr_sleep() to the rescue: we have provided a different implementation of a fine-grain sleep service within the Linux kernel, which also has the advantage of not requiring kernel recompilation. In fact, it can be mounted as an external Linux module[1]. Our solution is based on a single reference unit for the specification of the sleep period, namely the nanosecond. This allows us to retain the

[1] Our Linux patch is available at [36].

target sleep interval	1 μs		5 μs		10 μs		50 μs		100 μs		200 μs	
	mean	99p	mean	99p	mean	99p	mean	99p	mean	99p	mean	99p
`nanosleep()`	58.95	69.91	62.45	66.75	67.59	76.15	107.75	115.69	158.26	165.54	258.1	269.97
`hr_sleep()`	3.803	3.920	8.642	9.00	14.76	15.13	57.72	68.87	107.89	115.64	208.39	215.35

Table 1: Measured sleep period lengths (in μs) for the standard `nanosleep()` and our `hr_sleep()` for different values of the target sleep interval - mean and 99th percentile are shown.

same fine-grain capability of the original implementation. However, we did not rely on user-space memory for the specification; rather we used a single CPU register for passing the sleep-period information to the kernel level software. This allowed us to fully avoid user/kernel memory moves, thus completely bypassing the costs of the kernel level infrastructure that handles all the tasks associate with this type of cross-ring (user/kernel) data move, and the risk of being preemepted while executing the data move task. As an additional advantage from this bypassing, we better amortize the costs for managing the KPTI security patch used in Linux to fight hardware-speculation based attacks, like Meltdown [10]. In fact, when KPTI is active, the TLB is flushed upon kernel level access, because of the passage from the user level page table to the kernel level one (this requires rewriting the CR3 register on x86 processors). Hence, the access to the user space area keeping the information to be used by the conventional `nanosleep()` system call would lead to a TLB miss along the critical path of the thread execution, which is instead completely avoided in our solution. Additionally, having a single-value representation of the sleep period—as we do in our solution—allows avoiding the need for transforming the multi-value representation to the bitmask representing the timer expiration period into the corresponding entry of the timer wheel, as instead it is done by the conventional implementation of the `nanosleep()` service in the Linux kernel. This further helps saving CPU cycles (and also energy, as we will show via experimental data) and more promptly starting the actual sleep phase of the thread, thus increasing the precision of the actual sleep interval.

On the other hand, our new version of the kernel level software module implementing the sleep service, which we named `hr_sleep()`, is also based on the additional optimization of keeping the entry to be linked to the timer wheel on the already reserved kernel-level stack area for the calling thread. Hence, beyond speeding up the preamble phase of the sleep service, we also have the advantage of no interaction with other kernel level services (such as memory allocators) upon thread resume after the sleep phase. This again favors the avoidance of early preemptions—in fact several kernel level services are preemptable—once the thread takes control of the CPU. Overall, our implementation fully avoids (i) cross-ring data movement and the usage of the corresponding services, (ii) interaction with out of stack memory allocation.
A graphical representation of the main activities carried out by our version of the sleep service are depicted in Figure 1b.

3.2 Actual Thread Operations

In this section, we describe how threads in charge of processing packets operate in Metronome. To this end, let us start with a brief discussion of the state-of-the-art DPDK architecture: on the receiving side, NICs may convey their incoming traffic into a single Rx queue or either split such traffic into multiple Rx queues through RSS. A DPDK thread normally owns (and manages) one or more Rx queues, while an Rx queue belongs to (namely, is managed by) one DPDK thread [37]. Therefore, the behavior of a DPDK thread is no more than an infinite `while(1)` loop in which the thread constantly polls all the Rx queues it is in charge of. This approach rises some important shortcomings such as (i) greedy usage of CPU even in light load scenarios (a problem we already pointed to) and (ii) prevention of any Rx queue sharing among multiple threads. As for point (ii) we note that in 40GbE+ NICs, queues experience heavy loads despite the use of the RSS feature, e.g. on a 100Gb port with 10 queues, each queue can experience 10Gb rate traffic or even more. Preventing multi-threaded operations on each single Rx queue, and the exploitation of hardware parallelism for processing incoming packets from that queue, looks therefore to be another relevant limitation.

Compared to the above described classical thread operations in state of the art DPDK settings, we believe smarter operations can be put in place by sharing a Rx queue among different threads and putting these threads to sleep, when queues are idle, for a tunable period of time, depending on the current traffic characteristics. In other words, via a precise fine-grain sleep service, and lightweight coordination schemes among threads, we can still control and improve the trade-off between resource usage and efficiency of packet processing operations.

To this end, the fine-grain `hr_sleep()` service we presented in Section 3.1 has been coupled in Metronome with a multi-threaded approach to handle the Rx queues. In more detail, in our DPDK architecture we have multiple threads that sleep (for fine grain periods) and then, upon execution resume, race with each other to determine a single winner that will actually take care of polling the state of some Rx queue for processing its incoming packets. In this approach we do not rely on any additional operating system services to implement the race; rather, we implemented the race resolution protocol purely at user space via atomic Read-Modify-Write instructions, in particular the CMPXCHG instruction on x86 processors, which has been exploited to build a lightweight `trylock()` service. The race winner is the thread that atomically reads and modifies a given memory location (used as the lock associated with an Rx queue), while the others simply iterate on calling our new `hr_sleep()` service, thus immediately (and efficiently, given the reduced CPU-cycles usage of `hr_sleep()`) leaving the CPU—given that another thread is already taking care of checking with the state of the Rx queue, possibly processing incoming packets[2].

We also note that using multiple threads according to this scheme allows creating less correlated awake events and CPU-reschedules, leading to (i) more predictability in terms of the maximum delay we may experience before some Rx is checked again for incoming packets and (ii) less work to be done for each thread, since the same workload is split across more cores. This is true especially when

[2] Interested readers can have a look at Appendix A for a basic coding example of DPDK-traditional and Metronome approaches.

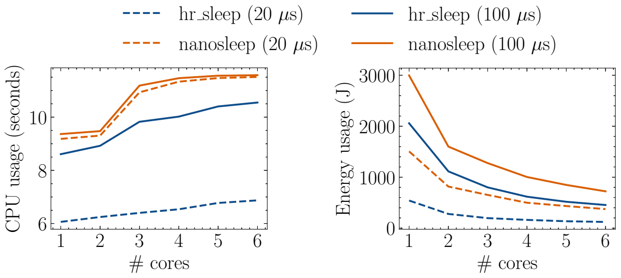
the CPU-cores on top of which Metronome threads run are shared with other workload. In fact, the multi-thread approach reduces the per-CPU load of Metronome. This phenomenon of *resiliency* to the interference by other workloads will be assessed quantitatively in Section 5.6, along with the benefits for the applications sharing the same cores with Metronome.

Overall, with Metronome we propose an architecture where Rx queues can be efficiently shared among multiple threads: to each queue corresponds a lock which grants access for the possessing thread to that queue. Threads can acquire access to a queue through our custom `trylock()` implementation, which provides non-blocking and minimal latency synchronization among them. For each of its queues, every thread tries to acquire the corresponding lock, and passes to the next queue if lock acquisition fails. Otherwise, if the thread wins the lock race it processes that queue as long as there are still incoming packets, then it releases the lock once the queue is idle. Once a thread has processed (or at least has tried to process) the Rx queues, it can go to sleep for a period of time proportional (and controllable in a precise fine-grain manner) to the traffic weight it has experienced during its processing. Scheduling an awake-timeout through our custom sleep service enables very precise and cheap thread-sleep periods, which are essential at 10Gb+ rates, and can still provide resource savings at lower rates. How a thread can elicit an awake-timeout period without incurring an Rx queue filling is carefully explained through our analytical model in Section 4. This model is used to make the Metronome architecture self-tune its operations, providing suited trade-offs between resource usage (CPU cycles and energy) and packet processing performance.

3.3 Assessment

In this section, we compare our new `hr_sleep()` to the standard Linux `nanosleep()` service in terms of both precision of the sleep period and resource (CPU and energy) usage by threads. The tests have been conducted on an isolated NUMA node equipped with Intel Xeon Silver 2.10GHz cores. The server is running Linux kernel 5.4.

3.3.1 Sleep Period Precision. To compare `hr_sleep()` with the standard Linux `nanosleep()` implementation, we run an experiment where a million samples of the wall-clock-time elapsed between the invocation of the sleep-service and the resume from the sleep phase are collected. This wall-clock-time interval has been measured via start/end timer reads operated through the `__rdtscp()` function. Table 1 shows the mean and 99th percentile for both the sleep services with different timer granularity, from 1 μs to 200 μs. These data have been collected by running the thread issuing the sleep request as a classical `SCHED_OTHER` (normal) priority thread—we recall that Metronome is devised with no need for imposing strict settings on the assignment of resources to DPDK threads so as to enable maximal flexibility by the platform administrator in CPU-sharing scenarios. We can observe the remarkable benefits of `hr_sleep()`, which proves to be deeply more precise in terms of average sleep time, compared to the requested sleep timeout, and definitely more reliable in terms of variance. While we clearly expected a strong advantage for very fine granularity of the requested sleep timeout (e.g., 1 μs, 5 μs...) we were surprised to experiment still a significant difference for much larger timeouts (e.g, 100-200 μs).



(a) CPU usage **(b) Energy usage**

Figure 2: `nanosleep` vs. `hr_sleep()` with 1M cycle iterations

3.3.2 CPU usage. We now compare the different levels of CPU usage when employing the classical `nanosleep()` service and our `hr_service()` in a Metronome-based DPDK architecture. The testbed DPDK application is a level 3 forwarder (further discussed in Section 5) which terminates after a million iterations inside a typical Metronome loop (see Listing 2 - Appendix A). We choose 20 and 100 μs as the fixed sleep-timeouts to be tested since they are good candidates for small and big sleep times respectively. Experiments are executed with thread count ranging from 1 to 6 (with thread pinned to different CPU-cores) and each sample comes out as the mean from 30 repetitions.

The CPU usage is retrieved from the main thread through the `getrusage()` service, by sampling it right before launching the slave threads and right after they end their execution. In order to only focus on the difference between the CPU usage by the two sleep services, tests are conducted under no traffic. Hence threads just use the CPU for issuing new sleep requests upon their execution resume. Figure 2a shows the results: for the same amount of work to be executed, our `hr_sleep()` service significantly decreases the CPU cycles needed, for the reasons we already explained (very cheap execution path and avoidance of preemption before the thread leaves the CPU or is CPU-rescheduled). Hence it looks to be a more appropriate (CPU efficient) solution for controlled timelines of thread activities. Furthermore, the maximal gain compared to `nanosleep()` is achieved via `hr_sleep()` with minimal sleep interval, namely 20 μs. Given that minimal sleep interval gives rise to increased likelihood that threads try to compete for accessing the Rx queue, this is an indication of the extreme effectiveness of the combination of `hr_sleep()` with the `trylock()` based race for making race-looser threads promptly leave the CPU. With longer sleep intervals, more likely threads find the Rx queue available for being locked, so that additional checks are attempted to verify the state of the queue, which (independently of the used sleep service) give rise to the usage of CPU cycles.

3.3.3 Energy usage. The very same test scenario is repeated for energy consumption measurements. Energy is retrieved from the main thread right before and after the slaves execution through the Intel RAPL package [38]. Figure 2b shows that `hr_sleep()` provides a significant gain also in energy saving: for 20 μs sleep periods, our service consumes a third of the energy amount required when relying on `nanosleep()`. In any case, `hr_sleep()` still provides large energy savings with the 100 μs sleep period.

4 METRONOME ADAPTIVE TUNING

In this section we provide an approach to adaptively tune the behavior of the Metronome architecture. Metronome is designed to operate via a sequence of *renewal cycles* $\Theta(i)$, which alternate *Vacation Periods* with *Busy Periods*. As shown in Figure 3, a *vacation period* $V(i)$ is a time interval where all the deployed packet-retrieval threads are set to *sleep mode*, hence incoming packets, labeled as $N_V(i)$ in the figure, get temporarily accumulated in the Receive (Rx) buffer. When the first among the sleeping threads wakes up (wins the race, via a successful `trylock()`, for handling the incoming packets from the Rx queue), a *busy period* $B(i)$ starts. This period will last until the whole queue is depleted by either the $N_V(i)$ formerly accumulated packets, as well as the new $N_B(i)$ packets arriving along the busy period itself $B(i)$ — see the example in Figure 3. Clearly, while performing the job, the thread can be CPU-descheduled by the OS for classical time-sharing reasons.

After depleting the queue, the involved thread will return to sleep. Note that other concurrent threads which wake up during a busy period will have no effect on packet processing—failing in the `trylock()` they will just note that Rx queue unloading is already in progress and will therefore instantly return to sleep, thus freeing CPU resources for other tasks.

4.1 Metronome Multi-Threading Strategy

As later demonstrated in Section 5.6, Metronome relies on multiple threads so as to guarantee increased robustness against CPU-reschedule delays of each individual Metronome thread, which is no longer in sleep state (the sleep timeout has fired and the thread was brought onto the OS run-queue). Such delay can be caused by CPU-scheduling decisions made by the OS—we recall that these decisions depend on the thread workload, their relative priorities and their current binding towards CPU-cores.

In such conditions, Metronome's control of the vacation period duration is not direct, as it would be in the single-thread case by setting the relevant timer, but it is *indirect* and stochastic, as this period is the time elapsing between the end of a previous busy period and the time in which some deployed thread awakes again and acquires the role of manager of the Rx queue. The question therefore is: *how to configure the awake timeouts of the different deployed Metronome threads*?

Unfortunately, the simplest possible approach of *equal* timeouts comes along with performance drawbacks: we will demonstrate later on in Figure 7 that when timeouts are all set to a same value, CPU consumption significantly degrades as load increases, which is antithetic with respect to the objectives of Metronome. Indeed, especially under heavy packet arrival rate, threads would wake-up, therefore consuming CPU cycles, just to find out that another thread is already doing the job of unloading the Rx queue!

We thus propose a **diversity-based** strategy for configuring the wake-up timeouts of different threads, which aims at mimicking a classical *primary/backup* approach, but without any explicit (and necessarily adding some extra CPU consumption) coordination, i.e. by using purely random access means. Each thread independently classifies itself as being in *primary* or *backup* state, according to the following rules:

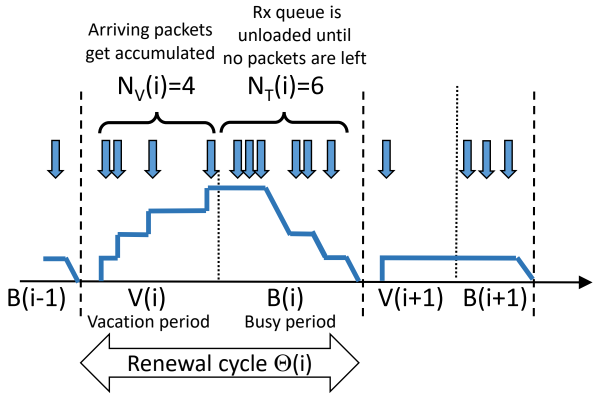


Figure 3: System model & renewal cycle

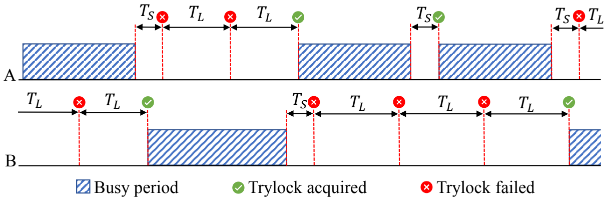


Figure 4: Vacation period and timeline of residual awake timeouts

- A thread becomes *primary* when it gets involved in a service time (it is the winner of the `trylock()` based race); at the end of the busy period it carried out, it reschedules its next wake-up time after a "short" time interval T_S;
- A thread classifies itself as *backup* when it wakes-up and finds an on going busy period (i.e. another thread is already unloading the queue); it then schedules its next wake-up time after a "long" time interval $T_L > T_S$.

In high load conditions, the above rules yield a scenario in which one thread at a time (randomly changing in the long term - see Figure 4) is in charge to poll the Rx queue at a reasonable frequency, whereas all the remaining ones occasionally wake up just for fall-back acquisition of the ownership on the Rx queue if for some reason the current primary gets delayed, e.g. by the OS CPU-scheduling choices. Conversely, at low loads more threads will happen to be simultaneously in the *primary* state, thus permitting to significantly relax the requirements on the "short" awake timeout T_S and motivating the adaptive strategy introduced in Section 4.3.

4.2 Metronome Analysis

4.2.1 Background. Let us non-restrictively assume that, once a thread wakes up, the packets accumulated in the Rx buffer get retrieved at a constant rate μ packets/seconds (this assumption is discussed in more details in Appendix B). It readily follows that the duration of the busy period $B(i)$ depends on the number of accumulated packets, and, more precisely, it comprises two components: i) the time necessary to deplete the first $N_V(i)$ packets arrived during $V(i)$, plus ii) the extra time needed to deplete the next $N_B(i)$ packets arrived since the start of the the busy period—in formulae:

$$B(i) = \frac{N_V(i) + N_B(i)}{\mu} \tag{1}$$

Since $N_V(i)$ and $N_B(i)$ depend on the vacation period $V(i)$, in most generality drawn from a random variable V, we can take conditional expectation at both sides of (1) with respect to V. Being λ the (unknown) mean packet arrival rate, we obtain the following fixed point equation[3] in $E[B|V]$:

$$E[B|V] = \frac{1}{\mu}E[N_V(i) + N_B(i)|V] = \frac{\lambda}{\mu}(V + E[B|V]). \tag{2}$$

which yields an explicit expression of how a busy period $E[B|V]$ is affected by the relevant vacation period:

$$E[B|V] = V\frac{\lambda/\mu}{1-\lambda/\mu} \tag{3}$$

If we conveniently define $\rho = \lambda/\mu$, we can derive an explicit expression which relates ρ to the controllable Vacation Period duration V and the relevant observable Busy Period $E[B|V]$— this expression will be indeed used to estimate ρ in Section 4.3:

$$\rho = \frac{E[B|V]}{V + E[B|V]} \tag{4}$$

4.2.2 Vacation Period statistics at high load. It is useful to start from two simplified mean-value analyses relying on two opposite set of assumptions valid at either high load or low load. The two different models will be then blended into a single one in Section 4.3. Let $M \geq 2$ be the number of deployed Metronome threads. In *high load conditions*, for reasons that will soon become evident, we can assume that only one of such threads is in the primary state, whereas all the remaining $M-1$ are in backup state. Once the primary thread releases the Rx queue lock and schedules its short timer T_S, two possible cases may occur:

- no backup thread wakes up during the sleep timeout T_S; in this case the primary thread will get back control of the Rx queue for the next round, and will remain primary;
- one of the remaining $M-1$ backup threads wake up *before* the end of the sleep timeout T_S and thus becomes primary; when the former primary thread wakes up, it will find a busy period[4] and will therefore acquire the role of backup thread, rescheduling its next wake up timeout after a time T_L.

Let us now make the assumption that the (current) $M-1$ backup threads were earlier CPU-rescheduled at independent random times. This *Decorrelation* assumption, indeed later on verified in Figure 5 using experimental results, is justified by the fact that each service time, due to its random duration, de-synchronizes the primary thread CPU-reschedule from the remaining ones; since after a few busy cycles all threads will have the chance to become primary, even if initially being CPU-scheduled at around the same times, their CPU-rescheduling instants will rapidly "decorrelate".

The statistics of the random variable V (vacation period) can be computed as the *minimum* between i) the fixed wake-up timeout T_S of the primary thread, and ii) the wake-up timeout of any of the remaining $M-1$ threads, which, owing to the previous decorrelation assumption, have been CPU-rescheduled in any random instant in the range $0, T_L$ before the end of the current busy period. It readily follows that the cumulative probability distribution function of V is given by:

$$CDF_V(x) = P(V \leq x) = \begin{cases} 1 - \left(1 - \frac{x}{T_L}\right)^{M-1} & x < T_S \\ 1 & x \geq T_S \end{cases} \tag{5}$$

and the mean vacation period for a given configuration of the short and long awake timeouts, and for a given number of threads, is trivially computed as:

$$E[V] = \int_0^{T_S} (1 - CDF_V(x))dx = \frac{T_L}{M}\left(1 - \left(1 - \frac{T_S}{T_L}\right)^M\right) \tag{6}$$

Finally, the probability that one of the $M-1$ backup threads gain access to the Rx queue at its wake-up time is given by

$$P_{s,succ} = \int_0^{T_S} \frac{1}{T_L}\left(1 - \frac{x}{T_L}\right)^{M-2} dx = \frac{\left(1 - \frac{T_S}{T_L}\right)^{M-1}}{M-1} \tag{7}$$

4.2.3 Vacation period statistics at low load. While, at high load, a neat pattern emerges in terms of one single primary thread at each time, with multiple backup threads, it is interesting to note that at low load Metronome yields a completely different behavior. Indeed, owing to equation (3), as the offered load reduces, the average busy period duration becomes small with respect to the vacation period duration. It follows that when a primary thread gets control of the Rx queue, it very rapidly releases such control, so that another thread waking up will find the queue available with high probability. It follows that in the extreme case, *all* threads will always remain in the primary state[5] and thus will periodically reschedule their next wake-up times after a short interval T_S. This case is even simpler to analyze than the previous one, as the CDF of the vacation time directly follows from (5) by simply setting $T_L = T_S$ and by considering M "competitors", in formulae:

$$CDF_V(x) = P(V \leq x) = 1 - \left(1 - \frac{x}{T_S}\right)^M \tag{8}$$

and mean vacation period simplifying to $E[V] = T_S/M$.

4.2.4 Experimental verification of the decorrelation assumption. To verify the validity of the decorrelation assumption used in the above models, Figure 5 compares the probability distribution function obtained by taking derivative of the CDF in equation (5), i.e., for $x < T_S$,

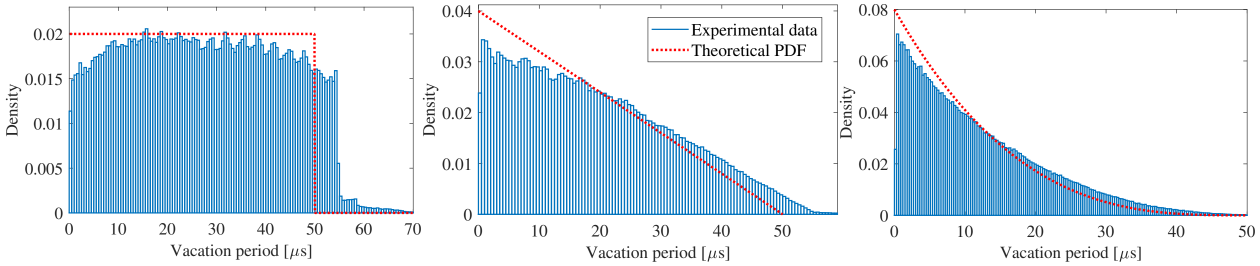
$$PDF_V(x) = \frac{M-1}{T_L}\left(1 - \frac{x}{T_L}\right)^{M-2} \tag{9}$$

with experimental results. We have specifically focused on the case $T_L = T_S$ as in this case the formula in equation 5 is expected to hold independently of the load (primary and backup threads use the same awake timeouts). Results, obtained with awake timeouts set to $50\mu s$ and different numbers of threads M, suggest that the decorrelation approximation is more than reasonable and the proposed model is quite accurate. Furthermore, the results also show that, in the real case, although rarely, actual CPU-reschedules after a sleep

[3] In the derivation, we exploited the following well known fact (direct consequence of the Little's Result): the average number $E[N]$ of packets arriving during a time interval of mean length $E[T]$ is $E[N] = \lambda E[T]$.

[4] In high load conditions, owing to equation 3, the average busy period lasts significantly longer than the vacation period.

[5] This is because each time an awaken thread finds the Rx queue not locked by another thread, then it acquires the primary role thanks to its successful `trylock()` operation.



(a) 2 cores **(b) 3 cores** **(c) 5 cores**

Figure 5: Vacation period PDF: analysis vs experiments, $T_S = T_L$

period can occur after the maximum time delay T_L, because of CPU-scheduling decisions by the OS—for example favoring OS-kernel demons. However, such impact becomes almost negligible in Metronome with just $M = 3$ deployed threads, pointing to the relevance of the adopted multi-threading approach.

4.3 Metronome Adaptation and Tradeoffs

Whenever the mean arrival rate is non-stationary, but varies at a time scale reasonably longer than the cycle time, the load conditions can be trivially run-time estimated using equation (4). For instance, the simplest possible approach is to consider for $\rho(i) = \lambda(i)/\mu$ the exponentially weighted estimator:

$$\rho(i) = (1-\alpha)\rho(i-1) + \alpha\frac{B(i)}{V(i)+B(i)} \tag{10}$$

Established that measuring the load is not a concern for Metronome, a more interesting question is to devise a mechanism which adapts the awake timeouts to the time-varying load. The obvious emerging trade-off consists in trading the polling frequency, namely the frequency at which threads wake up, with the duration of the vacation period which directly affects the packet latency. Indeed, if we assume that the serving thread is capable to drain packets from the Rx queue at a rate μ greater than or equal to the link rate, namely the maximum rate at which packets may arrive (in our experiments, 10 gigabit/s), then once the thread starts the service, packets will no longer accumulate delay. Therefore, the worst case latency occurs when a packet arrives right after the end of the last service period, and is delayed for an entire vacation period.

It follows that an adaptation strategy that *targets a constant vacation period duration irrespective of the load* appears to be a quite natural approach. Let us recall that, under the assumption $T_L >> T_S$, the average vacation period at high load, given by equation (6) simplifies to $E[V] \approx T_S$. Conversely, at low load, we obtained $E[V] = T_S/M$. Therefore, being $\bar{V}$ our target constant vacation period, the rule to set the timer T_S at either high or low loads reduces to:

$$\begin{cases} T_S = \bar{V} & \text{highload} \\ T_S = M \cdot \bar{V} & \text{lowload} \end{cases} \tag{11}$$

The analysis of the general case (intermediate load) is less straightforward, but can be still formally dealt with by assuming that threads are independent and are in primary or backup state according to the probability that, while they wake up, they find the Rx queue idle or busy, respectively. As showed in Appendix C, we can prove that, in this general case, under the assumption $T_L >> T_S$, the rule to set the timer T_S becomes:

$$T_S = M\frac{1-\rho}{1-\rho^M} \cdot \bar{V} \tag{12}$$

which, as expected, converges to (11) for the extreme high load case $\rho \longrightarrow 1$ and low load case $\rho \longrightarrow 0$.

Finally, we stress that Metronome does *not* sacrifice latency, but provides the *possibility to trade latency for CPU consumption*. Indeed, the duration of the chosen vacation period will determine the performance/efficiency trade-off: the longer the chosen vacation time, the lower the polling rate and thus the CPU consumption, at the price of a higher latency. If a deployment must guarantee low latency then it should either configure a small vacation time target, or even disable Metronome and use standard DPDK.

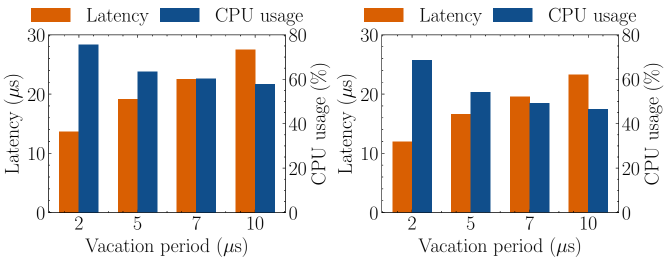
5 EXPERIMENTAL RESULTS

Our experimental campaign starts with the appropriate tuning for the $\bar{V}$, T_L and M parameters and the analysis of the subsequent tradeoffs. We then evaluate the differences between `hr_sleep()` and `nanosleep()` when used in Metronome in Section 5.2 and we test the adaptation capabilities of Metronome in Section 5.3. Section 5.4 discusses in detail both strengths and weaknesses of Metronome and DPDK in different aspects (latency, CPU usage and power consumption). Section 5.5 compares Metronome and XDP, while Section 5.6 shows Metronome's impact in common CPU sharing scenarios. For evaluating the system we used the testbed depicted in Section 3.3, running the `l3fwd` DPDK application [39] on an isolated NUMA node and generating traffic with MoonGen [40]. For benchmarking our system, we used the evaluation suite provided by Zhang et al. in [41]. Tests are done with 64B packets, as this is the worst case scenario[6]. Unless explicitly stated, tests in this section are executed using the `performance` CPU power governor and with parameters $\bar{V}$= 10 μs, T_L= 500 μs, M=3, each choice is motivated in the following section. For simplicity reasons, each NIC has only one Rx queue. Further tests for two different applications are showed in Figure 15.

[6] For tests regarding latency, since [41] uses Moongen's timestamping capabilities, it is necessary to add a 20B timestamp to the timestamped subset of packets, thus giving rise to a minimal difference in terms of offered throughput.

Target V [μs]	Measured V [μs]	Measured B [μs]	N_V	Loss (‰)
5	11.67	13.40	172.39	0
10	19.55	20.24	287.77	0
12	21.99	22.86	326.30	0.0037
15	26.23	27.25	385.18	0.023
20	33.28	38.32	494.39	1.180

Table 2: Mean busy and vacation period, N_V and packet loss for different target vacation periods.



(a) 10Gbps traffic

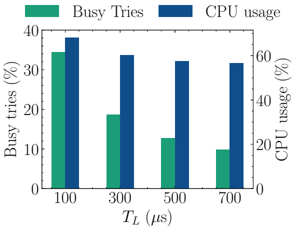
(b) 5Gbps traffic

Figure 6: Latency and CPU usage for different different target vacant times.

5.1 Parameters Tuning

First of all, we would like to find a vacation period $\bar{V}$ which permits us not to lose packets under line-rate conditions. Table 2 shows packet loss, vacation period and busy period for different values of $\bar{V}$, which represents the target V to be used when calling the `hr_sleep()` service: we found out that 10 μs is a good starting point as it provides no loss. The test was conducted using the suite's unidirectional p2p throughput test, as this test instantly increases the incoming rate from 0 to 14.88 Mpps, so as to be sure that this setting works even in the worst case scenario. We then analyzed the bidirectional throughput scenario by assigning 3 different threads to each Rx queue, as we found out that Metronome achieves the same maximum bidirectional throughput that DPDK can reach (11.61 Mpps per port) by constantly polling each Rx queue with a different thread. Once a good suitable minimum value for $\bar{V}$ is found, we investigate how tuning $\bar{V}$ affects CPU usage and latency: indeed, as Table 2 shows, the shorter $\bar{V}$, the less the queue is left unprocessed as the actual (namely, the measured) vacation time V decreases, so packets tend to experience a shorter queuing period. However, such an advantage does not come for free, as the CPU usage proportionally increases, as shown in Figure 6 for different traffic volumes. We note that all these tests have been performed by relying on 3 Metronome threads.

As for T_L, while letting backup threads sleep for a longer period of time alleviates the percentage of failed attempts of `trylock()` (busy tries), and therefore the number of wasted CPU cycles (as Figure 7 shows), a shorter T_L means higher reactivity when the primary thread is interfered by OS CPU-scheduling choices. For our evaluation we chose 500 μs as (i) it is 50 times bigger than the maximum T_S possible value, as our analytical model assumes that $T_L >> T_S$ (ii) Figure 7 shows that most of the advantage of increasing T_L happens before 500 μs, while between 500 and 700 μs we experimented a difference of only 1% in CPU usage and around 2% in busy tries. For our tests we choose 500 μs, as we are willing to sacrifice some CPU cycles for a better reactivity.



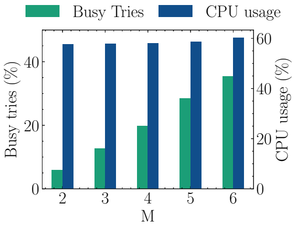
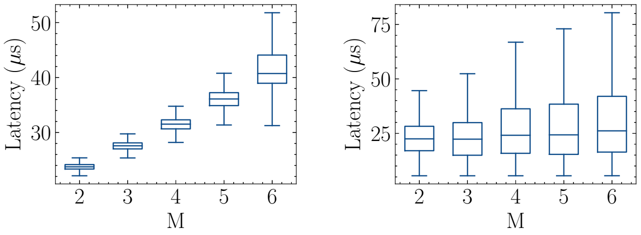


Figure 7: Busy tries and CPU usage versus T_L.

Figure 8: Busy tries and CPU usage versus M.

As for M, the philosophy underlying Metronome is the one of exploiting multiple threads for managing a Rx queue, but not the one using excessive (hence useless) thread-level parallelism. In fact, an excessive number of threads comes at almost no usefulness: Figure 8 shows how the percentage of busy tries increases linearly with the number of threads, along with a slight cost increase in terms of CPU usage. Furthermore, increasing the threads number comes along with a significant cost in terms of latency, as the more the threads, the more frequently a primary thread switches to the backup role leading to longer sleep periods as stated in Equation 12. We experimented considerable latency implications especially at high rates, as Figure 9a shows. Even for much lower rates, a substantial increase in variance is still visible (see Figure 9b). By the above hints, the evaluation is done with 3 threads.



(a) 10Gbps traffic

(b) 1Gbps traffic

Figure 9: Latency vs. the number of threads M

5.2 Comparing `hr_sleep()` and `nanosleep()`

Once we've found suitable values for Metronome on our testbed, we now focus running Metronome with the standard `nanosleep()` service rather than our `hr_sleep()` one, in order to underline the consequent performance implications. As Table 3 shows, a first experiment running Metronome with the same vacation period $\bar{V}$ that achieved no packet loss in the previous section witnessed a sensible loss (around 6%). Even by increasing the RX queue size to the maximum (4096 descriptors), `nanosleep()`' reduced accuracy still caused a sensible loss percentage. The limitations of such an approach were still visible (around 0.8% of loss) when reducing the vacation parameter to a very small one (1 μs).

This test clearly outlines that using `nanosleep()` for an adaptive packet processing is not feasible on 10Gbps links. Moreover, we ran some latency tests in order to underline the difference in terms of resolution for both sleep services. Figure 10 shows for different

RX queue size	$\bar{V}$ (μs)	Packet loss (%)
1024	10	6.166
2048	10	4.08
4096	10	3.893
4096	1	0.845

Table 3: Using `nanosleep()` in Metronome causes packet loss. In the same scenarios, `hr_sleep()` achieves no packet loss (10Gbps traffic with 64B packets)

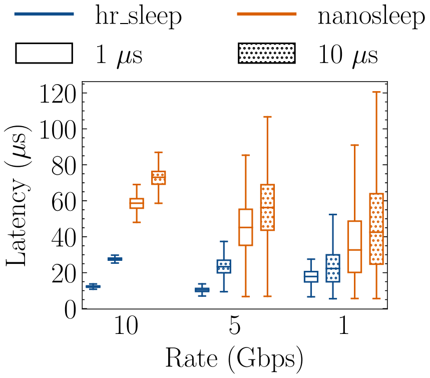


Figure 10: Latency boxplots for `hr_sleep()` and `nanosleep()`

throughputs[7], the latency boxplots with a fine grained timeout (1 μs, empty boxplots) and a more relaxed one (10 μs, dots-filled boxplots): the difference in resolution is pretty clear and further motivates the use of `hr_sleep()`.

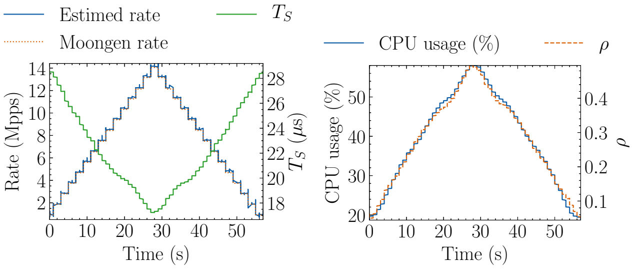
5.3 Adaptation

To test Metronome's dynamic capabilities to adapt to a varying offered load, we modified the Moongen `rate-control-methods.lua` example to generate constant bit rate traffic at a variable speed: in a time interval of 60 seconds, Moongen increases the sending rate every 2 seconds until a rate of 14 Mpps is reached at about 30 s, and then it starts decreasing. Figure 11a shows how Metronome *perfectly* matches the Moongen generated traffic rate and how the T_S parameter set by the threads proportionally adapts. Figure 11b proves that Metronome promptly adapts CPU usage with respect to the incoming traffic, starting from about 20% with no traffic and increasing up to 60% under almost line rate conditions. Also the ρ parameter correctly adjusts its value along with the traffic load.

5.4 Comparing Metronome and DPDK

We now focus on the comparison between the adaptive Metronome capabilities and the static, continuous polling mode of DPDK in terms of (i) induced latency, (ii) overall CPU usage and (iii) power consumption.

Latency: we tested Metronome in order to investigate how the sleep&wake approach impacts the end-to-end latency. One of our goals was to experiment a constant vacation period, therefore a constant mean latency. Figure 12a shows how Metronome (blue boxplots) successfully fulfills this requirement, despite a negligible increase under line-rate conditions, which seems obvious. DPDK clearly benefits from its continuous polling operations as it induces about half of the mean latency that Metronome does and is also more reliable in terms of variance (see Figure 12a, orange boxplots). However, rather than very low latency, Metronome targets an adaptive and fair usage of CPU resources with respect to the actual traffic. The minimum latency that Metronome can induce is mainly limited by two aspects: the first one is the Tx batch parameter. Since DPDK transmits packets in a minimum batch number which is tunable, as our system periodically experiments a vacation period some packets may remain in the transmission buffer for a long period of time without actually being sent: this is clearly visible as variance at low rates increases. To overcome such a limitation, we ran another set of tests with the transmission batch set to 1, so that no packets can be left in the Tx buffer. We found out positive impacts both on variance and (slightly) mean values for very low rates. Downgrading the Tx threshold to 1 comes at the cost of a 2-3% increase in CPU utilization at line rate. The second aspect is the minimum granularity that `hr_sleep()` can support, even if the sleep time requested is much smaller than microseconds (i.e., some nanoseconds). By tuning the first parameter and patching the `hr_sleep()` in order to immediately return control if a sub-microsecond timer is requested, we managed to obtain a 7.21 μs mean delay in Metronome which is very close to the DPDK minimum one (6.83 μs), and also a significant decrease in variance (0.62 μs in Metronome vs. 0.43 μs in DPDK) while still maintaining a 10% advantage in CPU consumption.



(a) Rate and T_S estimation

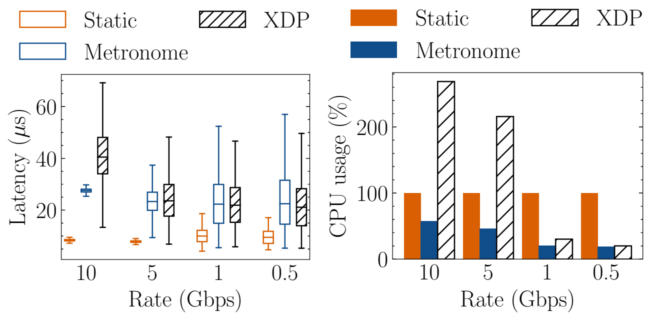
(b) CPU usage and ρ

Figure 11: Metronome's correct adaptation to the incoming traffic load

Total CPU usage: Figure 12b shows the significant improvements by Metronome (blue bars): while DPDK's greedy approach (orange bars) gives rise to fixed 100% CPU utilization, Metronome's adaptive approach clearly outperforms DPDK as it is able to provide 40% CPU saving even under line-rate conditions, while under low rate conditions the gain further rises to more than 5x (Metronome achieves around 18.6% CPU usage at 0.5Gbps). We underline that Metronome's CPU consumption could be further decreased by increasing the T_L value as explained in Section 5.1.

Power consumption: as for energy efficiency, it is critical to examine the two approaches depending on the different power governors available in Linux. More specifically, we concentrated on the two most performing ones, namely `ondemand` and `performance`. The first can operate at the maximum speed possible, but dynamically adapts the CPU frequency depending on the current load, while the second one works always at the maximum possible rate. Of course, while the former permits some power saving at the cost of increased

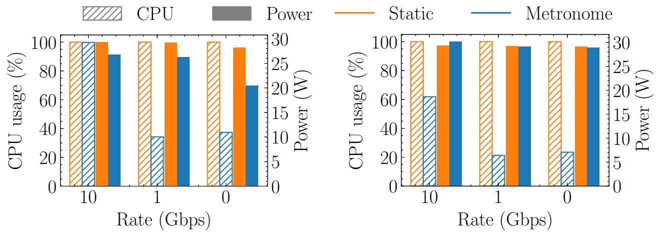
[7] despite the considerable loss caused by normally using `nanosleep()` with a 10Gbps throughput, the use of a large RX size (4096) and a higher packet size (70B) allowed us not to lose packets. Tests with `hr_sleep` are always done with 64B packets.



(a) Latency boxplot

(b) CPU usage

Figure 12: L3 Forwarder example running static DPDK, Metronome and XDP



(a) `ondemand`

(b) `performance`

Figure 13: Power vs CPU utilization for different power governors.

CPU occupancy, the latter wastes more power but lowers the CPU usage. This trade-off is clearly visible in Figures 13a and 13b: except for the 10Gbps throughput under the `performance` power governor scenario, Metronome achieves less power consumption than polling DPDK does, with the maximum gain reached when operating under no traffic with the `ondemand` governor (around 27%). We underline that in the `ondemand` scenario Metronome's CPU usage is higher than in the previously seen plots. While we concentrated on the `performance` governor since we wanted to minimize Metronome's CPU consumption, these tests show that depending on the user/-provider's needs, Metronome can also achieve significant power saving when compared to static polling DPDK.

5.5 Comparing Metronome and XDP

We believe it is the case for Metronome to be also compared against XDP [23]: this work has a similar motivation to Metronome's main one (reduced, proportional CPU utilization) and is nowadays integrated into the Linux kernel. Despite this similar goal, the approach of the two architectures is quite different: XDP is based on interrupts and every Rx queue in XDP is associated to a different, unique CPU core with a 1:1 binding. Through a conversation with one of the XDP authors on GitHub [42], we discovered that our Intel X520 NICs (running the `ixgbe` driver) achieve at their best a close-to line-rate performance: in fact, the maximum we managed to get is 13.57 Mpps with 64B packets. To do this, we had to equally split flows between four different cores running the `xdp_router_ipv4` example (the most similar one to DPDK's `l3fwd`). The graphs now discussed are obtained using the minimal number of cores for XDP in order not to lose packets[8] (4 cores on 10Gbps and 5Gbps, 1 core on 1Gbps and 0.5Gbps). We remark that if XDP is deployed with the goal of potentially sustaining line-rate performance, on our test server it should statically be deployed on four cores since there's no way to dynamically increase the number of queues (and therefore, cores) without the user's explicit command through `ethtool`: in that case, XDP's total CPU usage increases at 52% @1Gbps and 34% @0.5Gbps. Figure 12a shows the latency boxplot for XDP: while (even with interrupt mitigation features enabled) we see an increased latency at line rate, we experimented similar latencies at lower rates (we underline that decreasing Metronome's $\bar{V}$ and Tx batch parameter we could obtain lower latency results as shown in Section 5.4, while XDP is already operating at its best performance). Figure 12 shows XDP's mean total CPU utilization, which is clearly much higher because of the per-interrupt housekeeping instructions required to lead control to the packet processing routine, which have an incidence especially at higher packet rates. On the other hand, XDP occupies no CPU cycles at all under no traffic, while Metronome still periodically checks its RX queues. This different approach permits Metronome to be highly reactive in case of packet burst arrivals (as showed in Section 5.1), while XDP loses some tens of thousands of packets in this case before adapting.

5.6 Impact

Finally, we analyze Metronome's capabilities to work in a normal CPU sharing scenario, where different tasks compete for the same CPU. We first focus on motivating our multi-threading approach, then we show that the CPU cycles not used by Metronome can be exploited to run other tasks in the meantime without significantly affecting Metronome's capabilities. In both experiments, Metronome is sharing its same three cores with a VM running `ferret`, a CPU-intensive, image similarity search task coming from the PARSEC [43] benchmarking suite. Because the Metronome task is more time sensitive than the `ferret` one, we give Metronome a slight scheduling advantage by setting its nice value to -20, while the VM's niceness is set to 19 since it has no particular time requirements. In any case, the two are still set to belong to the same `SCHED_OTHER` (normal) priority class.

The case for multiple threads: While we previously stated that a few threads are better for Metronome, we now clarify the reason for using multiple threads by scheduling the VM running the `ferret` program on one core. When running Metronome on the same single core, because of the CPU conflicting scenario the maximum throughput achievable by `l3fwd` is around 8 Mpps. If we deploy Metronome on three cores (one of these three cores is the same used by the VM), only one thread will be highly impacted by the CPU-intensive task and therefore will unlikely act like a primary thread. In this case `l3fwd` achieves no packet loss on a 10Gbps link, and the same scenario happens if we schedule the same VM running `ferret` on two of the three cores shared with Metronome. The next paragraph shows that also when all of the three Metronome threads are (potentially) impacted by `ferret`, they can still forward packets at line rate, thanks to the reduced likelihood that all of them (once brought back to the runqueue after the sleep period) are impacted

[8] We decreased the Mpps sending rate to 13.57 by sending 72B packets, so that XDP wasn't losing packets.

simultaneously because of the decisions of the OS CPU-scheduler. These experiments clearly show that running Metronome on multiple threads leads to improved robustness against common CPU sharing scenarios and interference by other workloads.

Co-existing with other tasks: we now demonstrate that Metronome's sleep&wake approach enables the CPU sharing of other tasks without major drawbacks, while DPDK's static, constant polling approach denies such possibilities. We first ran `ferret` on one core, both without any concurrent task and with a static DPDK polling `l3fwd` application on the same core. Then, we scheduled `ferret` on three cores and the three Metronome threads on the same cores. As Figure 14 shows, sharing the CPU with a static polling task causes `ferret` to almost triple its duration, while Metronome's multi-threading and CPU sharing approach only causes a 10% increase. Moreover, standard DPDK's single core approach couldn't keep up with the incoming load, achieving a maximum of 7.31 Mpps, while Metronome achieved no packet loss even when all of its three cores were shared with a CPU intensive program such as `ferret` (see Table 4). We underline that Metronome's multi-threading strategy implies that the same workload is shared between multiple threads, thus the more the cores, the less the work every thread needs to perform and therefore the more they can co-exist with other tasks without affecting performances, as this test shows.

	alone	w/ `ferret`
static DPDK	14.88	7.34
Metronome	14.88	14.88

Table 4: Throughput (Mpps) for static DPDK and Metronome

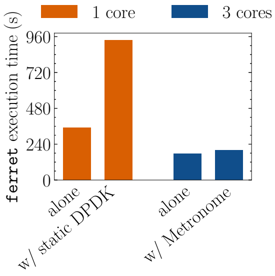


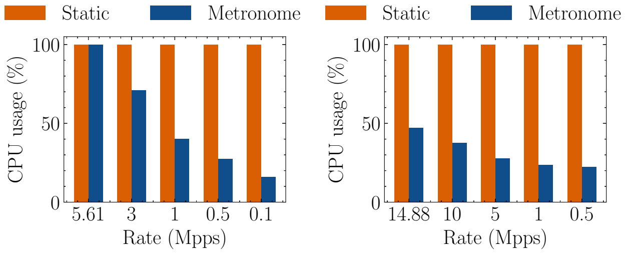
Figure 14: Execution time for `ferret`

5.7 Tested applications

To further assess the flexibility and the wide breadth of Metronome, we show three DPDK applications that we successfully adapted to the Metronome architecture, namely two DPDK sample applications as a L3 forwarder [39] and an IPsec Security Gateway [44], as well as FloWatcher-DPDK [13], a high-speed software traffic monitor. They are available at [36].

L3 forwarder The `l3fwd` sample application acts as a software L3 forwarder either through the longest prefix matching (LPM) mechanism or the exact match (EM) one. We chose the LPM approach as it is the most computation-expensive one between the two. We have used the `l3fwd` application to exhaustively test Metronome's performances in Section 5, so we refer the readers to that Section for further performance implications.

IPsec Security Gateway This application acts as an IPsec end tunnel for both inbound and outbound network traffic. It takes advantage of the NIC offloading capabilities for cryptographic operations, while encapsulation and decapsulation are performed by the application itself. Our tests perform encryption of the incoming packets through the AES-CBC 128-bit algorithm as packets are later sent to the unprotected port. The DPDK sample application achieves a maximum outbound throughput of 5.61 Mpps with 64B packets in static polling mode: once we adapted the application to Metronome's algorithm, we found out that we were able to reach the exact same throughput. In fact, one of Metronome's threads was always processing packets and therefore never releasing the trylock shared with the other threads, this is clearly visible in Figure 15a. For lower rates, Metronome clearly outperforms the static approach as rates get lower.



(a) IPsec Security Gateway

(b) FloWatcher-DPDK

Figure 15: CPU usage

FloWatcher-DPDK FloWatcher is a DPDK-based traffic monitor application providing tunable and fine-grained statistics, both at packet and per-flow level. FloWatcher can either act through a run to completion model or a pipeline one: we chose the former since the receiving thread is also calculating the statistics, therefore providing a more challenging scenario for Metronome. We find out that Metronome provides the same performances that the static DPDK approach does in terms of zero packet loss and correct statistics calculation, while reaching major improvements in CPU utilization, as Figure 15b shows a 50% gain even under line rate traffic and almost a 5x gain with 0.5 Mpps traffic.

6 CONCLUSIONS

This paper has proposed and assessed Metronome, an approach devised to replace the continuous and CPU-consuming DPDK polling with a sleep&wake, load-adaptive, intermittent packet retrieval mode. Metronome was technically enabled by our development of a new high resolution timed-sleep Linux service, named `hr_sleep()`, way more precise and efficient than the classical `nanosleep()`, with common thread priorities. Metronome's viability has been assessed by integrating it into three different common DPDK applications, and by showing its significant improvements primarily in terms of CPU utilization (and, partially, also in terms of power consumption), and therefore ability to release precious CPU cycles to business applications. We finally stress that such gains do not come for free, but are traded off with an extra latency toll, which therefore needs to be taken into account, and carefully configured using the tuning knobs provided by our approach, when (and if) considering the usage of Metronome with time-critical applications.

Acknowledgments: We thank our anonymous reviewers as well as our Shepherd for their valuable feedbacks which helped us improve this paper. We also thank Giuseppe Siracusano and Sebastiano Miano for helping us in tuning XDP.

Listing 1: A standard DPDK polling loop

```
1  while (1) {
2    for (i = 0; i < n_queues; i++) {
3      nb_rx = receive_burst(queue[i], pkts, BURST_SIZE);
4      if (nb_rx == 0)
5        continue;
6      process_and_send_pkts(pkts, nb_rx);
7    }
8  }
```

Listing 2: Our novel DPDK processing loop

```
1  while (1) {
2    lock_taken = false;
3    for (i = 0; i < n_queues; i++) {
4      if(!trylock(lock[i]))
5        continue;
6      lock_taken = true;
7      while(nb_rx = receive_burst(queue[i], pkts,
           BURST_SIZE))
8        process_and_send_pkts(pkts, nb_rx);
9      unlock(lock[i]);
10   }
11   if (lock_taken)
12     hr_sleep(timeout_short);
13   else
14     hr_sleep(timeout_long);
15 }
```

APPENDIX

A SKELETON CODE

We briefly compare the classical and Metronome methods through a simplified (yet meaningful) code example of a typical DPDK thread routine. This example only focuses on the different coding approaches, rather than other aspects (e.g., implementing the actual network functionalities, calculating the optimal timer through our analytical model...). Both examples show a typical packet processing task. The usual DPDK implementation is shown in Listing 1, while our novel proposal is depicted in Listing 2. While both solutions include a set of Rx queues to be processed, in Listing 1 those Rx queues are exclusively managed by a specific thread, while in Listing 2 queues are shared among multiple threads and therefore require access through the `trylock()` mechanism (see line 4). In Listing 1 for each of its queues, the thread tries to retrieve a burst of packets (line 3) of maximum size `BURST_SIZE`, processes it (line 6) and immediately scans again its set of queues, regardless of the fact that those queues may be experiencing low traffic (or no traffic at all). We highlight that this behavior is the real cause of the 100% constant CPU utilization by a single thread, as threads are working in a *traffic-unaware* manner. As for the later point, this level of CPU usage is negatively reflected on energy consumption and also on turbo-boost waste.

Listing 2 shows our novel approach: once the lock for a certain queue is acquired, the thread processes that queue until it becomes empty (`while()` loop in lines 7-8) and then releases the lock. If a certain lock can't be granted, that queue is skipped (`continue;` at line 5) as a different thread is already processing it. When the queues scanning if finished, the thread checks if it has taken the lock (line 11) or not (line 13), then the thread can call our `hr_sleep()` service as it can go to sleep for a `timeout_short` in the first case and for `timeout_long` for the latter. Furthermore, as we discussed in Section 4, being our `hr_sleep()` precise and cheap, model based tuning of the `timeout` can be effectively put in place to achieve suited resource-usage vs performance trade-offs. Despite the simplicity of these examples, we believe they clearly point out the difference between a *traffic-aware* policy and a static one simply based on greedy resource usage.

B CAVEATS AND DETAILS

In this Appendix we discuss some supplementary technical details at the basis of our assumptions. We specifically start from the assumption used in the model presented in Section 4.2: packet retrieval rate μ independent on the packet size. Even if not strictly necessary[9], our assumption of μ constant and independent of the packet size is actually motivated by the specific way in which DPDK handles packets. Indeed, DPDK does not process packets by physically moving them from the NIC, but it just moves the relevant descriptors which populate the Rx queue.

Since a typical DPDK application consists of a loop where a receive function is executed at each iteration, the service rate can somewhat be influenced by: (i) the loop length (that is, how much time passes between two consecutive receive operations) and, (ii) how many descriptors are processed by the receive function in each cycle. DPDK usually processes descriptors in a batch defining the maximum number of packets to be processed at each invocation. Usually, this value is set to 32 as it provides a nice tradeoff for the batching benefits without affecting latency. Some interference on the rate μ may also be inducted by OS interrupts or because of preemption of DPDK threads by some higher priority thread (like an OS kernel demon). However, the multi-tread approach taken by Metronome is devised just to make DPDK more resilient towards this kind of interference scenario, and with no need for dedicated resources—as said, one of our targets is to make DPDK effective in CPU-sharing contexts. In fact, nowadays OS kernels (like the Linux kernel) adopt temporary (if not fixed) binding approaches of threads to CPUs—with periodic migration of threads across the CPU-cores for load balancing. Hence, having multiple Metronome threads that can become primary while managing the NIC decreases the likelihood that all the DPDK threads (statically pinned to different CPU-cores at startup time) share their CPU-core with higher priority interfering threads. On the other hand, we have already mentioned that Metronome—including its `hr_sleep()` architectural support—is devised with no need to explicitly impose high priority to its threads. This leaves extreme flexibility to the infrastructure owner in terms of resource-usage configuration. The transmitting process is also influenced by batching, as DPDK moves descriptors to a transmit queue only if a certain batch threshold is reached for the same amortization reasons. This doesn't directly affect the retrieval rate, but can rather influence the latency that DPDK induces. Transmission and receiving queues permit the host CPUs to dialog with NICs through the DMA technology: such queues usually have a

[9] The renewal arguments brought about in this section remain valid if we replace deterministic quantities with their mean - in other words even if we consider the alternative model of constant retrieval rate in terms of a constant rate of C *bits per second*, opposed to μ *packets per second*, we would just need to set $\mu = E[P]/C$, with $E[P]$ being the average packet size.

variable length (on an Intel X520 NIC, users can choose a Rx/Tx queue length between 32 and 4096 descriptors).

C METRONOME'S ADAPTATION POLICY UNDER GENERAL LOAD CONDITIONS

In this Appendix we propose a simplified, but still theoretically motivated, approach which allows us to blend the results obtained via the two extreme low and high load models into a single and convenient analytical framework.

More specifically, in *intermediate load conditions* we cannot anymore assume that just one single thread (as in high load conditions), or all threads (as in low load conditions), are in primary state. Rather, a part from the *single* thread that has last depleted the Rx queue, and which is therefore surely in primary state, also *some* of the remaining $M-1$ threads will be in primary state whereas others will be in backup state. Let us therefore introduce a random variable P which represents the number of the remaining threads in primary state. $M-1-P$ will therefore be the number of remaining threads in secondary state.

Let us now assume that each of the remaining $M-1$ threads can be *independently* found in primary or backup state with probability p (which will be determined later on). Then, the random variable P representing the number of remaining threads in primary state trivially follows the Binomial distribution:

$$\mathrm{Prob}(P=k) = \binom{M-1}{k} p^k (1-p)^{M-1-k}$$

Then, we can compute the average vacation time also in intermediate load conditions, by taking conditional expectation over this newly defined random variable P. This permits us to generalize equation (6) as follows:

$$E[V] = E[E[V|P]] =$$

$$= \sum_{k=0}^{M-1} \binom{M-1}{k} p^k (1-p)^{M-1-k} \int_0^{T_S} \left(1 - \frac{x}{T_S}\right)^k \left(1 - \frac{x}{T_L}\right)^{M-1-k} dx =$$

$$= \int_0^{T_S} \left(1 - \frac{px}{T_S} - \frac{(1-p)x}{T_L}\right)^{M-1} dx =$$

$$= \frac{1 - ((1-p)(1-T_S/T_L))^M}{M\left(\frac{p}{T_L} + \frac{1-p}{T_S}\right)}$$

Furthermore, assuming $T_L >> T_S$, we can conveniently simplify the above expression and approximate it as:

$$E[V] = \frac{T_S}{M} \cdot \frac{1-(1-p)^M}{p} \tag{13}$$

Note that, for $p \longrightarrow 0$, namely when the probability to find another thread in the primary state becomes zero (high load conditions), equation (13) converges to the expected value T_S, whereas $E[V] = T_S/M$ for $p = 1$ (as per low load conditions, i.e. all threads becoming primary).

As a last step, it suffices to relate p with the offered load. To this purpose, let $\rho = \lambda/\mu$ be the probability that the Rx queue is busy at a random sample instant. It is intuitive to set $p = (1-\rho)$, as the probability p that a thread is in the primary state is the probability that when this thread has last sampled the queue, it has found it idle, i.e. $1-\rho$. This finally permits us to formally support our proposed formula (12) as load-adaptive T_S setting strategy. Summarizing for the reader's convenience, being $\bar{V}$ a constant target vacation period, and ρ the current load estimate, T_S can be set as:

$$T_S = M\frac{1-\rho}{1-\rho^M} \cdot \bar{V}$$

Note that this rule can be conveniently rewritten in a more intuitive and simpler to compute form, as

$$T_S = M\frac{1-\rho}{1-\rho^M} = \bar{V}\frac{M}{1+\rho+\cdots+\rho^{M-1}}$$

REFERENCES

[1] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE ANCS*. IEEE.

[2] Zhifeng Xu, Fangming Liu, Tao Wang, and Hong Xu. Demystifying the energy efficiency of network function virtualization. In *2016 IEEE/ACM 24th Int. Symp. on Quality of Service (IWQoS)*.

[3] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proc. of the 10th European Conference on Computer Systems*, EuroSys 2015.

[4] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229.

[5] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM SIGCOMM Conference*, pages 123–137.

[6] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[7] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. Building a power-proportional software router. In *2012 USENIX Annual Technical Conference*, pages 89–100, 2012.

[8] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Commun. of the ACM*, 62(2):48–60, 2019.

[9] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore. Reconfigurable Network Systems and Software-Defined Networking. *Proceedings of the IEEE*, 103(7): 1102–1124, 2015.

[10] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium*, 2018.

[11] Jeff Barr. Elastic Network Adapter – High Performance Network Interface for Amazon EC2. URL https://aws.amazon.com/it/blogs/aws/elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/.

[12] Laxmana Rao Battula. DPDK (Data Plane Development Kit) for Linux VMs now generally available. URL https://azure.microsoft.com/en-us/blog/dpdk-data-plane-development-kit-for-linux-vms-now-generally-available/.

[13] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi. FloWatcher-DPDK: Lightweight Line-Rate Flow-Level Monitoring in Software. *IEEE Transactions on Network and Service Management*, 16(3):1143–1156, 2019.

[14] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference*, pages 101–112.

[15] N. Bonelli, S. Giordano, and G. Procissi. Network Traffic Processing With PFQ. *IEEE Journal on Selected Areas in Communications*, 34(6):1819–1833, 2016.

[16] PF_RING ZC (Zero Copy). URL https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[17] DPDK. URL https://www.dpdk.org/.

[18] Gorrie, L. et al. Snabb: Simple and fast packet networking. URL https://github.com/snabbco/snabb.

[19] Data Plane Development Kit Power Optimization on Advantech* Network Appliance Platform. Technical report, Intel, 2015. URL https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/dpdk-power-optimization-advantech-white-paper.pdf.

[20] Xuesong Li, Wenxue Cheng, Tong Zhang, Jing Xie, Fengyuan Ren, and Bailong Yang. Power Efficient High Performance Packet I/O. In *Proceedings of the 47th Int. Conf. on Parallel Processing*, 2018.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2018)*, pages 51–66, 2018.

[22] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, August 1997.

[23] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT 2018, pages 54–66.

[24] XDP supported drivers, . URL https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp.

[25] DPDK - Supported NICs, . URL http://core.dpdk.org/supported/nics/.

[26] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*, pages 361–378.

[27] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proc. of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341.

[28] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*, pages 33–46, Boston, MA.

[29] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, pages 93–109.

[30] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[31] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the 2017 ACM SIGCOMM Conference*, pages 404–417.

[32] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 ACM SIGCOMM Conference*, pages 30–43.

[33] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2012)*, pages 253–266.

[34] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[35] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the 2017 ACM SIGCOMM Conference*, page 29–42.

[36] Metronome: adaptive packet retrieval in DPDK. https://github.com/marcofaltelli/Metronome.

[37] Muthurajan Jayakumar. Data Plane Development Kit (DPDK)—Multicores and Control Plane Synchronization . URL https://software.intel.com/content/www/us/en/develop/articles/dpdk-data-plane-multicores-and-control-plane-synchronization.html.

[38] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), 2018.

[39] Sample Applications User Guides - L3 Forwarding Sample Application. URL https://doc.dpdk.org/guides-19.11/sample_app_ug/l3_forward.html.

[40] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proc. of 2015 Internet Measurement Conf.*, IMC '15, pages 275–287.

[41] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, Luigi Iannone, and James Roberts. Comparing the Performance of State-of-the-Art Software Switches for NFV. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments And Technologies*, CoNEXT 2019, pages 68–81.

[42] Achieving line rate with xdp_fwd using Intel X520 #53. URL https://github.com/xdp-project/xdp-project/issues/53.

[43] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[44] Sample Applications User Guides - IPsec Security Gateway Sample Application. URL https://doc.dpdk.org/guides-19.11/sample_app_ug/ipsec_secgw.html.