Metronome: Adaptive and Precise Intermittent Packet Retrieval in DPDK

Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi

Abstract—The increasing performance requirements of modern applications place a significant burden on software-based packet processing. Most of today’s software input/output accelerations achieve high performance at the expense of reserving CPU resources dedicated to continuously poll the Network Interface Card. This is specifically the case with DPDK (Data Plane Development Kit), probably the most widely used framework for software-based packet processing today. The approach presented in this paper, descriptively called Metronome, has the dual goals of providing CPU utilization proportional to the load, and allowing flexible sharing of CPU resources between I/O tasks and applications. Metronome replaces DPDK’s continuous polling with an intermittent sleep&wake mode, and revolves around a new multi-threaded operation, which improves service continuity. Since the proposed operation trades CPU usage with buffering delay, we propose an analytical model devised to dynamically adapt the sleep&wake parameters to the actual traffic load, meanwhile providing a target average latency. Our experimental results show a significant reduction of the CPU cycles, improvements in power usage, and robustness to CPU sharing even when challenged with CPU-intensive applications.

Index Terms—Computers and information processing, network function virtualization, data centers, cloud computing.

I. 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%. Fully dedicating a CPU, the most greedy component in terms of power consumption [7], [8], 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 [9], [10], 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 [11]).

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 [12], [13] 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 [14], an approach devised to replace the continuous DPDK polling with a sleep&wake intermittent approach. Albeit this might seem in principle an obvious idea, its advantages are linked to several factors that we cope with in this article. First, a suited implementation/usage of sleep&wake operating system services seems in principle an obvious idea, its advantages are linked to several factors that we cope with in this article. First, a suited implementation/usage of sleep&wake operating system services
supported by either the Linux `nanosleep()` service or our own new service called `hr_sleep()` [14]. The latter offers a few advantages and is also independent of limitations related to system parameterization and thread priorities. Second, Metronome revolves around a novel architecture and operating mode for DPDK, where incoming traffic, from either a single receive queue or multiple ones, is shared between multiple threads—as we will discuss this also offers advantages by the side of robustness versus operating system thread-scheduling decisions. These threads dynamically switch—in a coordinated manner—from polling the receive queue to sleep 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.

Overall, the contributions we provide in this article can be summarized as follows:

- exploiting a fine grained sleep&wake service—in particular the `hr_sleep()` service—we define the Metronome multi-threaded architecture for DPDK applications, based on extremely low thread-coordination overhead. It boils down CPU usage compared to classical DPDK settings, and offers a better capability to exploit hardware level parallelism. As an indirect effect, Metronome positively impacts energy efficiency under specific workloads;
- we present an analytical model for Metronome, which is used for driving allocation of CPU to threads, making the DPDK framework dynamically adapt its behavior (and its demand for resources) to the workload;
- we extensively assess Metronome on 10 Gbps NICs, in various load conditions, and we test its integration in three different applications: L3 forwarding, IPsec, and FloWatcher [15], a high-speed software traffic monitor;
- we extend the evaluation of Metronome to 40 Gbps NICs, where multiple receive queues need to be used and therefore, orchestrated by the Metronome algorithm.

Metronome is publicly available at [16].

II. BACKGROUND AND 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 [17], PFQ [18] and PF_RING ZC [19]) or bypass the kernel through a user space driver, like DPDK [20] and Snabb [21]. Such frameworks rely on high performance, batch transferring mechanisms such as DMA and zero copy [19], 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 [22] 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 [23], 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 downgrading of the clock frequency reduces 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 fully 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 [24]. 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 control at fine grained 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 based on an analytical model, that allows taking runtime decisions depending on 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 [9], [10] 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 interrupts, packet copies to user space and so on. Moreover, an interrupt-based architecture operating at extreme interrupt arrival speed may cause livelocks [25]. 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. There is a growing interest in XDP [26], [27], 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 match DPDK’s performances ( [26] and Sec.V-D) and functionalities, it also currently supports fewer drivers than DPDK does [28], [29]. Our solution is instead fully integrated with DPDK. Works like Shenango [30] and Zygos [31] have a different target, namely latency sensitive user space applications, like memcached or RPCs. The former uses a static polling datapath for dispatching packets and orchestrating core reallocations for the above applications while the latter is a specialized operating system for handling a large number of network connections efficiently. Other contributions try to accelerate packet processing by moving computation to modern NICs [32], [33], [34], [35].
III. Metronome Architecture

A. Fine-Grained Thread Sleep Service

The precision of the thread-sleep interval supported by the operating system, 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 to 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-grained) sleep periods of threads is based on the nanosleep() system call. A few limits of this service are related to its dependency on a slack factor assigned to threads, which is checked when they request to sleep. This factor can be controlled using the prctl() system call, putting it to the minimal value of 1. If such setting is not adopted, then for any thread that is not in the real-time CPU-scheduling class we have at least 50 microseconds as the slack imposed by the Linux kernel, which makes the awake of the thread less controllable in terms of precision under fine-grained sleep requests. Furthermore, when entering the kernel level execution of nanosleep(), the Thread Control Block (TCB) is checked because of the need to determine the current slack value, which makes the service run a few machine instructions to reconcile the real value to be adopted in the sleep phase with the information kept in the TCB.

While developing Metronome, we also implemented an alternative sleep service, namely hr_sleep(), whose details are provided in [14]. This variant fully avoids any interaction with thread management information at kernel level (such as the current slack value kept in the TCB). Hence, it also avoids running the additional machine instructions needed to manage this information, for any CPU-scheduling class of threads (real-time or not). We refer the reader to [14] for an extended evaluation of this implementation. In any case, in Figure 1 we show that the performance of hr_sleep() is completely similar to the one of nanosleep() under the scenario where the latter is configured with the minimal admissible slack currently supported by the Linux kernel. As said, this setting of kernel-level parameters is not requested at all when using hr_sleep(). The tests have been conducted on an isolated NUMA node equipped with Intel Xeon Silver 2.1 GHz cores. The server is running Linux kernel 5.4. We have 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 __rdtscp(). We show the boxplots for both the sleep services with different timer granularity requests (1, 10, 100 μs). These data have been collected by running the thread issuing the sleep request as a classical SCHED_OTHER (normal) priority thread and—as hinted before—with the timer slack of nanosleep() set to 1μs. The results show that hr_sleep() provides some minimal gains both in terms of mean latency and variance even under such extreme setting of the slack value for nanosleep(). In any case, the avoidance of the reliance on kernel-level parameters makes hr_sleep() fully independent of any kernel configuration choice for the minimal admissible slack.

B. 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 split such traffic into multiple Rx queues through Receive Side Scaling (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 [36]. 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 raises 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 (see Section V-E).

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-grained 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 hr_sleep() service 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 grained periods) and then, upon execution resumption, 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. 1

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 the CPU-cores on top
of which Metronome threads run are shared with other work-
load. 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 V-E, 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
(see Figure 2): to each queue corresponds a lock which
grants access to that queue. Threads can acquire access to
a queue through our custom trylock() implementation,
which provides non-blocking and minimal latency synchro-
nization among them. For each of its queues, every thread tries
to acquire the corresponding lock, and passes to a different
queue chosen randomly (see Section IV-E) 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 to (and
controllable in a precise fine-grained manner depending on)
the traffic it has experienced during its processing. Scheduling
an awake-timeout through a fine-grained 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 IV.
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.

IV. 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 buffer. For simplicity,
we first consider a single Rx queue, then we expand our
model to multiple queues in Section IV-E. When the first
among the sleeping threads wakes up and 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
element in Figure 3.

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.

A. Metronome Multi-.Threading Strategy

As later demonstrated in Section V-E, Metronome relies
on multiple threads 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 (see Section II in the Appendix) 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:

- 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 thread that was 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 IV-D.

B. Metronome Analysis

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 $\mu$ packets/seconds (this assumption is discussed in more details in Appendix II). It readily follows that the duration of the busy period $B(i)$ depends on the number of accumulated packets, and, more precisely,
find a busy period\(^3\) 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 between 0 and \(T_L\) before the end of the current busy period. It readily follows that the cumulative probability distribution function of \(V\) is given by:

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

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 gains access to the Rx queue at its wake-up time is given by:

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

3) Vacation Period Statistics at Low Load: While, at high load, a neat pattern emerges in terms of one single primary thread at any 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\(^4\) 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) 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 period can occur after the maximum time delay \(T_L\), because of CPU-scheduling decisions by the OS—for example favoring OS-kernel daemons. However, such impact becomes almost negligible in Metronome with just \(M\) = 3 deployed threads, pointing to the relevance of the adopted multi-threading approach.

C. Adaptation Policy Under General Load Conditions

We propose a simplified, but still theoretically motivated, approach which allows us to blend the results obtained via

\(^3\)In high load conditions, owing to equation 3, the average busy period lasts significantly longer than the vacation period.

\(^4\)This is because each time an awakened thread finds the Rx queue not locked by another thread, then it acquires the primary role thanks to its successful trylock{} operation.
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 along time. Rather, apart from the single thread that has last depleted the Rx queue, 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:

\[
\text{Prob}(P = k) = \binom{M-1}{k} p^k (1-p)^{M-1-k}
\]

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:

\[
E[V] = E[E[V|P]] = \\
= \sum_{k=0}^{M-1} \binom{M-1}{k} p^k (1-p)^{M-1-k} \int_0^{T_S} \left( 1 - \frac{x}{T_S} \right)^k \left( 1 - \frac{x}{T_L} \right)^{M-1-k} dx = \\
\int_0^{T_S} \left[ \left( 1 - \frac{p x}{T_S} - \frac{(1-p)x}{T_L} \right)^{M-1} \right] dx = \\
= \frac{1}{\left( (1-p) \left( 1 - \frac{T_S}{T_L} \right) \right)^M} \\
\]

Furthermore, assuming \( T_L \gg T_S \), we can conveniently simplify the above expression and approximate it as:

\[
E[V] = \frac{T_S}{M} \cdot \frac{1 - (1-p)^M}{p} \tag{10}
\]

Note that, for \( p \to 0 \), namely when the probability to find another thread in the primary state becomes zero (high load conditions), equation (10) converges to the expected value \( T_S \), whereas \( E[V] = T_S / M \) for \( p = 1 \) (as for low load conditions, i.e. all the 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 (13) as the load-adaptive \( T_S \) setting strategy. Summarizing for the reader’s convenience, being \( \bar{V} \) a constant target vacation period, and \( \rho \) 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:

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

D. 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:

\[
\rho(i) = (1 - \alpha) \rho(i-1) + \alpha \frac{B(i)}{V(i) + B(i)} \tag{11}
\]

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 \( \mu \) greater than or equal to the link rate, namely the maximum rate at which packets may arrive (in our single-queue 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 \( T_L \gg T_S \), 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:

\[
T_S \begin{cases} 
= \bar{V} & \text{highload} \\
= \frac{M \cdot \bar{V}}{\rho} & \text{lowload}
\end{cases}
\tag{12}
\]

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 shown in Section IV-C, we can prove that, in this general case, under the assumption \( T_L \gg T_S \), the rule to set the timer \( T_S \) becomes:

\[
T_S = M \frac{1 - \rho}{1 - \rho^M} \cdot \bar{V}
\tag{13}
\]

which, as expected, converges to (12) for the extreme high load case \( \rho \to 1 \) and the extreme low load case \( \rho \to 0 \).

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.

E. The Multiquaue Case

When Metronome is used with 40+Gb NICs, one queue becomes not enough to sustain line rate traffic. Therefore, a split of the incoming traffic into multiple receive queues through RSS is needed. We now introduce the \( N \) parameter, which represents the number of Rx queues for a certain NIC. Given \( M \) as the total number of threads in the system, we believe it should be at least as big as \( N \), so that every queue can have one primary thread associated to it (\( M \geq N \)). In this scenario, we have \( N \) primary threads (since everyone of them has won the lock race for a different queue) and \( M - N \) secondary threads. In a scenario with multiple queues, we believe it is not efficient to statically bind a thread to a certain queue (see Section V-E.6), so we propose a different approach:

- once a primary thread has won the race and depleted a queue, it goes to sleep for a \( T_s \) period and when it wakes up, it contends for the same queue as we know it is likely for it to win the race again.
- once a backup thread has lost a lock race, it chooses the queue to be contended at its next wakeup randomly.

The random selection of the next queue for the backup thread ensures a certain decorrelation among the threads in the next queue selection and also fairness with respect to the queue checks. While the \( T_L \) value remains fixed, we update equation (13) as follows:

\[
T_s = \frac{M}{N} \cdot \frac{1 - \rho_i}{1 - \frac{M}{N}} \cdot \bar{V} \text{ for } i = 1, \ldots, N \quad (14)
\]

We notice two differences with the single queue version. The former is that the \( M \) parameter is now replaced with \( M/N \), as that is the average number of threads taking care of a certain queue at any moment. The latter is that the \( \rho \) parameter is now per-queue based, as each queue can experience different traffic rates (and therefore, queue occupancy and vacation periods) at any time.

V. EXPERIMENTAL RESULTS

Our experimental campaign starts with the appropriate tuning\(^5\) for the \( V \) and \( M \) parameters and the analysis of the subsequent tradeoffs. Section V-B shows how to convert \( V \) in terms of latency. Section V-C discusses in detail both strengths and weaknesses of Metronome and static DPDK in different aspects (latency, CPU usage and power consumption). Section V-D compares Metronome and XDP, while Section V-E shows the impact of Metronome in common CPU sharing scenarios. While tests up to Section V-E have been conducted with a single Rx queue (using Intel X520 NICs), Section V-F evaluates Metronome in a multi-queue scenario (with Intel XL710s). For evaluating the system we used a server running Linux kernel 5.4 equipped with

\(^5\) Tuning for the \( T_L \) parameter is available in the Appendix, as well as a traffic adaptation test.

| Target \( V \) (\( \mu s \)) | Measured \( V \) (\( \mu s \)) | Measured \( B \) (\( \mu s \)) | \( N_V \) | Loss (%) |
|---|---|---|---|---|
| 5 | 11.67 | 13.40 | 172.39 | 0 |
| 10 | 19.55 | 20.24 | 287.71 | 0 |
| 15 | 21.99 | 22.86 | 326.30 | 0.037 |
| 20 | 26.23 | 27.25 | 385.18 | 0.023 |

Intel® Xeon® Silver @2.1 GHz, running the \texttt{13fwcl} DPDK application [37] on an isolated NUMA node and generating constant bit rate UDP traffic with MOOGen [38]. For benchmarking our system, we used the evaluation suite provided by Zhang et al. [39], as well as the Intel RAPL package [40] and the \texttt{getrusage()} syscall to retrieve energy usage and CPU consumption. Tests are done with 64B packets, as this is the worst case scenario.\(^6\) Unless explicitly stated, the tests are executed using the performance CPU power governor and with parameters \( V = 10 \mu s, T_L = 500 \mu s, M = 3 \)—each choice is motivated in the following section. Further tests for two different applications are also shown in the Appendix.

A. Parameters Tuning

First of all, we would like to find a vacation period \( V \) which permits us not to lose packets under line-rate conditions. Table I shows packet loss, vacation period and busy period for different values of \( V \), which represents the target \( V \) to be used when calling the \texttt{hr_sleep()} service: we found out that 10 \( \mu s \) 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 \( V \) is found, we investigate how tuning \( V \) affects CPU usage and latency: indeed, as Table I shows, the shorter \( V \), 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 \( M \), the philosophy underlying Metronome is the one of exploiting multiple threads for managing a Rx queue, not the one using excessive (hence useless) thread-level parallelism. In fact, an excessive number of threads comes at the expense of busy tries increases linearly with the number of threads, along with a slight cost increase in terms of CPU usage.

\(^6\) For tests regarding latency, since [39] uses MOOGen’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.
as to tune the system for a desired mean latency requirement. Being $N$ the number of packets in our system, it follows that (using the same notation and arguments of Section IV-B)

$$E[N] = E[N_V] + E[N_B] = \lambda E[V] + \rho E[N] \rightarrow E[N]$$

Therefore, from Little’s result, we obtain:

$$E[T] = \frac{E[V]}{1 - \rho} \quad (15)$$

Since a direct measure of “just” the queueing delay is technically cumbersome, Figure 8 compares the measured end-to-end latency (box plots) with the results of the above formula (green line), for various measured mean vacation periods $E[V]$ (in turns obtained with different load values $\rho$). Apart from the constant offset due to the propagation delay (orange dashed line) which is not included in the analytical results, these do closely match experimental ones in terms of slope of the resulting plots.

### C. 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.

1) 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 10a 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 achieves and is also more reliable in terms of variance (see Figure 10a - 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 on both 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 hr\_sleep() in order to immediately return control if a sub-microsecond sleep timeout is requested, we managed to obtain a 7.21 $\mu s$ mean delay in Metronome which is very close to the DPDK minimum one (6.83 $\mu s$), and also a significant decrease in variance (0.62 $\mu s$ in Metronome).
when operating under no traffic with the ondemand the traditional DPDK does, with the maximum gain reached scenario, Metronome achieves less power consumption than 10Gbps throughput under the off is clearly visible in Figures 11a and 11b: except for the this permits some savings in terms of power. This trade-cores need more time to get to the maximum speed, but on under line-rate conditions, while under low rate conditions the gain further rises to more than 5× (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 V-A.

3) Power Consumption: As for energy efficiency, it is critical to examine the two approaches depending on the different power governors [41] available in Linux. More specifically, we concentrated on the two most performing ones, namely ondemand and performance. The first can operate at the maximum possible speed, but dynamically adapts the CPU frequency by periodically examining the current CPU load and depending on some threshold values, while the second one keeps the CPU cores at their maximum speed while executing code. While ondemand permits a more adaptive CPU policy, it is less reactive than performance. In particular, CPU cores need more time to get to the maximum speed, but this permits some savings in terms of power. This trade-off is clearly visible in Figures 11a and 11b: except for the 10Gbps throughput under the performance power governor scenario, Metronome achieves less power consumption than the traditional 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 (∼90% at 10Gbps, ∼70% under no load). 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.

D. Comparing Metronome and XDP

We believe it is the case for Metronome to be also compared with XDP [26]: 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 (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 10a 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 V and the Tx batch parameter we could obtain lower latency results as shown in Section V-C, while XDP is already operating at its best performance). Figure 10 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 shown in Section V-A), while XDP loses some tens of thousands of packets in this case before adapting. In terms of power consumption (Figure 11) for the same reason discussed above XDP consumes more power when processing at line rate, while permits significant gains when no traffic is being processed.

E. 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 the 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 niceness 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.

1) 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 (when requiring to be brought back to the runqueue after the sleep period) are impacted 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.

2) Vacation Period Impact: In the Appendix, we show the same test performed in Section IV-B.2, Figure 5, with the addition of ferret interfering Metronome.

3) Latency Impact: Figure 12a shows the latency boxplots for Metronome with the ferret interference (blue) and without it (orange). The impact of interference on latency is that, once Metronome releases the CPU and consequently awakens, it will wait for some time before being rescheduled since the VM is not preempted immediately by the OS. This phenomenon of course happens less frequently under high load, as Metronome has more work to do and therefore is less likely to release the CPU, while it is more likely under low load, where the increase in latency is indeed more visible.

4) Co-existence 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, 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 12b 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 II). 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.

F. Going Multiqueue

Our evaluation now focuses on the multiqueue case analyzed in Section IV-E: tests have been conducted for both Metronome and static DPDK using Intel XL710 40Gbps NICs. These devices are limited by a maximum processing rate of 37Mpps [44]. In all tested environments, Metronome always reached the desired 37Mpps forwarding throughput. Traffic is distributed equally among the Rx queues through RSS, while in a later subsection we will discuss the unbalanced traffic case. We found out that the main components to be tuned for achieving the best performances in Metronome (assuming a fixed $V = 15\mu s$) are the number of Rx queues, the CPU power governor and the number of threads.

1) Tuning the Number of Queues: We test our l3fwd application using 2,3 and 4 Rx queues for the same 37Mpps throughput. Results for CPU and power consumption are available in Figure 13. We now focus on the performance power governor (Figures 13a,b,c), as we discuss the impact of the on-demand power governor in the next paragraph. The $\rho$ parameter and the busy tries percentage are also shown (see Figure 14) in order to better explain the results. As with 2 Rx queues every queue is experiencing high load...
traffic (∼18Mpps each), most of the time the queues are busy ($\rho = 0.7$ with 2 threads) and the CPUs are running at their maximum, so the main gain is in the CPU occupancy (150% with 2 threads, 156% with 8). While in the cases with many threads Metronome uses more power than static DPDK (here represented with dotted lines), it does not make much sense to use more than 4 threads to contend just two queues, as also the linear increase in busy tries (blue-filled bars in Figure 14a) suggests. When using a larger number of queues (3 or 4), the lower per-queue load permits Metronome to increase its gain compared to static DPDK both in CPU and power (see Figure 13c). In order to determine the number of queues to use, it is worth noticing that on one hand with a larger number of queues, $\rho$ decreases and, consequently, the number of busy tries decreases, which makes the Metronome algorithm more efficient. On the other hand, as the number of queues increases, so does the number of threads to deploy and consequently, power consumption. We believe that $\rho \sim 0.5$ is a good compromise, which in our experiments is reached with 3 Rx queues.

2) **Power Governors Matter:** While in the previous paragraph we focused on the performance governor, we now discuss the ondemand one, the difference between the two is explained in Section V-C. Figure 13d shows the results with 2 Rx queues: the initial decrease in power consumption is motivated by the fact that while with 2 threads, these can only be in the primary state, when increasing the number of threads, they tend to be backup ones (and therefore, to sleep for more time) because of the high percentage of busy tries (see the red-filled bars in Figure 14a). This is in turn caused by the steep increase of $\rho$ with the number of threads: since the CPU cores can execute at slower rates, threads will likely take more time to unload their Rx queues and therefore these will be busy for longer periods. This phenomenon is still visible with 3 queues and slightly with 4 queues. As the number of queues increases, the difference between the two power governors in terms of queue occupation $\rho$ and busy tries still remains significant but also slightly decreases (see the subfigures in Figure 14). Overall, the ondemand power governor permits to trade some extra CPU time for a better power efficiency:
also in this case the best advantages are visible with a larger number of queues. This further demonstrates Metronome’s capability to adapt to a lower per-queue load.

3) Tuning the Number of Threads: After presenting Figure 14, we can now draw some implications about the number of threads to be used in the multiqueue case. Our goal is to deploy a large enough number of threads (at least as big as the number of Rx queues, see Section IV-E) without incurring in too much busy tries (quantitatively, an upper bound of 10% of the tries), assuming the use of the performance power governor. We can see from Figure 14 that this is approximately achieved when the number of cores doubles the number of Rx queues. Therefore, we propose $N \leq M \leq 2N$ as a reasonable choice and we use this approach also in the later sections. In the supplementary material we show how this approach scales well also in 100Gb link scenarios.

4) Scaling to the Actual Traffic: Figure 15a shows the CPU consumption for Metronome and DPDK under different traffic rates, from 0 to line rate on an Intel XL710 (37Mpps). The test is done with 4 Rx queues with both Metronome and DPDK, and with $M = 5$ and $V = 15\mu s$ for Metronome. Our approach saves more than half of static DPDK’s CPU cycles while maintaining the same line-rate throughput, and improving even more at lower rates. Also in terms of power consumption (see Figure 15b), Metronome provides around 2-3W of advantage even when using a highly expensive power governor such as performance.

5) Unbalanced Traffic: We test Metronome’s multiqueue capabilities by continuously sending at line rate an unbalanced pcap file. The file is composed by 1000 packets, 30% of the packets belongs to the same UDP flow, while the other 70% is randomly generated and therefore equally split among the queues. In the test we use 3 Rx queues (without losing packets), so the most stressed queue processes around 53% of the total throughput, while the other two queues are in charge of 23% each. Table III shows some meaningful statistics for the test. Queue #2 is the most stressed. Therefore, it has the highest busy tries percentage and also the highest queue occupation $\rho$. It is worth noticing that, on a 3-minute test, queue #2 experienced less than half of the lock tries of queues #1 and #3: this trend validates the assumption in Section IV-A, where a busy queue tends to have only one primary thread at a time while a less occupied one is more likely to have more threads in the primary state simultaneously, and therefore, more tries.

6) Thread-to-Queue Binding Policy: We show that our random binding policy of a thread to a certain queue leads to increased resiliency with respect to the static binding one.
its ability to release precious CPU cycles to business applications. In the future, deploying Metronome in multi-NICs scenarios with per-NUMA node thread pools and a per-queue multithreaded Rx driver could be considered as further steps. We finally stress that such gains are traded off with an extra latency toll, which can be taken into account and configured using the tuning knobs provided by our approach, especially when (and if) considering the usage of Metronome with time-critical applications.

Acknowledgment

The authors would like to thank Giuseppe Siracusano and Sebastiano Miano for helping them in tuning XDP.

References

[1] S. Gallemüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, “Comparison of frameworks for high-performance packet IO,” in Proc. ACM/IEEE ANCS, May 2015, pp. 29–39.

[2] Z. Xu, F. Liu, T. Wang, and H. XU, “Demystifying the energy efficiency of network function virtualization,” in Proc. IEEE/ACM IWQoS, Jun. 2016, pp. 1–10.

[3] A. Verma, L. Pedrosa, M. Korupolu, A. C. Snoeren, and G. Carle, “Inside the network’s (datacenter) network,” in Proc. ACM SIGCOMM, Aug. 2015, pp. 123–137.

[4] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in Proc. ACM IMC, 2010, pp. 267–280.

[5] L. Nicosini, G. Rannacino, S. Ratnasamy, J. Chandrasekhar, and L. Rizzo, “Building a power-proportional software router,” in Proc. USENIX ATC, 2012, pp. 89–90.

[6] L. A. Barroso, U. Hölze, and P. Ranganathan, “The Datacenter as a Computer: Designing Warehouse-Scale Machines,” 3rd ed. San Rafael, CA, USA: Morgan & Claypool, 2018, ch. 1.

[7] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” Commun. ACM, vol. 62, no. 2, pp. 48–60, Jun. 2019.

[8] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, “Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads,” in Proc. USENIX NSDI, 2019, pp. 361–378.

[9] G. Prekas, M. Kogias, and E. Bugnion, “ZygOS: Achieving low tail latency for microsecond-scale networked tasks,” in Proc. ACM SOSP, 2017, pp. 325–341.

[10] B. Stephens, A. Akella, and M. Swift, “Loom: Flexible and efficient NIC packet scheduling,” in Proc. USENIX NSDI, 2019, pp. 33–46.

[11] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, “Enabling programmable transport protocols in high-speed NICs,” in Proc. USENIX NSDI, 2020, pp. 93–109.

[12] S. Han, K. Jang, A. Panda, S. Pulkar, D. Han, and S. Ratnasamy, “SoftNIC: A software NIC to augment hardware,” Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UC/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[13] G. Bianchi et al., “XTRA: Towards portable transport layer functions,” IEEE Trans. Netw. Serv. Manag., vol. 16, no. 4, pp. 1507–1521, Dec. 2019.

[14] M. Jayakumar. Data plane development kit (DPDK)—Multicores and control plane synchronization. Intel. Accessed: May 12, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/dpdk-data-plane-multicores-and-control-plane-synchronization.html.

[15] Sample Applications User Guides—L3 Forwarding Sample Application. Accessed: May 12, 2021. [Online]. Available: https://doc.dpdk.org/guides-19.11/sample_app Ug/l3_forward.html.

[16] P. Emmerich, S. Gallemüller, D. Raumer, F. Wohlfart, and G. Carle, “MoonGen: A scriptable high-speed packet generator,” in Proc. ACM IMC, Oct. 2015, pp. 275–287.

[17] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, “Comparing the performance of state-of-the-art software switches for NFV,” in Proc. ACM CoNEXT, Dec. 2019, pp. 68–81.

[18] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “RPL in action: Experiences in using RPL for power measurements,” ACM Trans. Model. Perform. Eval. Comput. Syst., vol. 3, no. 2, pp. 1–26, Jun. 2018.

[19] CPU Frequency and Voltage Scaling Code in the Linux Kernel. Accessed: May 12, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[20] Achieving Line Rate With XDP_FWD Using Intel X520 #53. Accessed: May 12, 2021. [Online]. Available: https://github.com/xdp-project/xdp-project/issues/53.

[21] L. Gorrie. SNABB: Simple and Fast Packet Networking. Accessed: May 12, 2021. [Online]. Available: https://github.com/snabbco/snabb.

[22] Intel. (2015). Data Plane Development Kit Power Optimization on Advantech” Network Appliance Platform. Accessed: May 12, 2021. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/dpdk-power-optimization-advantech-white-paper.pdf.

[23] X. Li, W. Cheng, T. Zhang, J. Xie, F. Ren, and B. Yang, “Power efficient high performance packet I/O,” in Proc. ACM ICPP, Aug. 2018, pp. 1–10.

[24] D. Firestone et al., “Azure accelerated networking: SmartNICs in the public cloud,” in Proc. USENIX NSDI, 2018, pp. 51–66.

[25] J. C. Mogul and K. Ramakrishnan, “Eliminating receive livelock in an interrupt-driven kernel,” ACM Trans. Comput. Syst., vol. 15, no. 3, pp. 217–252, 1997.

[26] T. Holland-Jørgensen et al., “The express data path: Fast programmable packet processing in the operating system kernel,” in Proc. ACM CoNEXT, Dec. 2018.

[27] M. S. Brunella et al., “xDPD. Efficient software packet processing on FPGA NICs,” in Proc. USENIX NSDI, 2020, pp. 973–990.

[28] XDP Supported Drivers. I/O Visor Project. Accessed: May 12, 2021. [Online]. Available: https://github.com/ovs/bcc/blob/master/docs/kernel-versions.md#xdp.
Marco Faltelli is currently pursuing the Ph.D. degree with the University of Rome “Tor Vergata.” His research activities combine computer networking and computer architecture to design high-performing and scalable architectures for both network hardware and software solutions. His research has been awarded a Microsoft Research Ph.D. Fellowship. In the past, he collaborated with CNIT on the 5G-PICTURE European project. He further collaborates with Axbryd, a startup company founded by the NetProg Group, University of Rome “Tor Vergata.”

Giacomo Belocchi is currently pursuing the Ph.D. degree in computer science with the University of Rome “Tor Vergata.” His research interests focus on programmable dataplane languages and platforms, NF acceleration, and NFV composition. He has been working in research since 2017, contributing to large European Research projects, such as H2020 5G-Picture. He is the Co-Founder and main Software Engineer at Axbryd Inc., a research spin-off startup disrupting how networks are deployed and operated by using eBPF and FPGAs.

Francesco Quaglia received the M.S. degree in electronic engineering and the Ph.D. degree in computer engineering from the Sapienza University of Rome in 1995 and 1999, respectively. He has worked as an Assistant Professor and an Associate Professor with the Sapienza University of Rome from September 2000 to June 2017. Since then, he works as a Full Professor at the University of Rome “Tor Vergata.” His research interests include parallel and distributed computing systems and applications, operating systems, high performance computing, and cyber-security.

Salvatore Pontarelli received the master’s degree in electronic engineering from the University of Bologna, Bologna, Italy, in 1999, and the Ph.D. degree in microelectronics and telecommunications from the University of Rome “Tor Vergata,” Rome, Italy, in 2003. He participated in research projects funded by public bodies (FP7 and H2020) and private companies. He collaborated with various manufacturers of Internet network devices (Cisco, USA and Mellanox Technologies, USA). He is currently an Assistant Professor with the Department of Computer Science, Sapienza University of Rome, Rome. His main research interests are the design of high-speed hardware architectures for programmable network devices, the implementation of hash-based data structures (bloom filters and cuckoo tables), and network dataplane programmability.

Giuseppe Bianchi is currently a Full Professor in networking at the University of Roma “Tor Vergata.” His research activities include wireless networks (an area where he has carried out pioneering research work on WLAN modeling and assessment), programmable network systems, security monitoring and vulnerability assessment, traffic modeling and control, and is documented in about 300 peer-reviewed international journals and conference papers, accounting for more than 20,000 citations (source: Google Scholar). He has coordinated six large scale EU projects, and has been (or still is) editor for several journals in his field, including IEEE/ACM Transactions on Networking, IEEE Transactions on Wireless Communications, IEEE Transactions on Network and Service Management, and Computer Communications (Elsevier).