A Programmable Hardware Calendar for High Resolution Pacing

Salvatore Pontarelli[1], Giuseppe Bianchi[1,2], Michael Welzl[3]
[1]Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy.
[2]University of Rome Tor Vergata, Italy.
[3]Department of Informatics, University of Oslo, Norway.

***Abstract*—The challenge addressed in this paper consists in offloading packet-based pacing to a hardware Network Interface Card, while retaining the flexibility of software timers. In this direction, we propose, design, implement, and evaluate a hardware calendar, which can be programmed via a simple yet very flexible programming interface leveraging stateful (adaptive) per-packet timers. We show, for both specific examples (exponential, linearly increasing, etc) as well as for the general case, how to derive such a per-packet timer setting from a high-level desired *rate envelope*. Further, we describe and evaluate an FPGA implementation which relies on a *novel* insertion strategy for solving collisions in the calendar's hash table.**

I. Introduction

With a growing understanding of how important it is to minimize latency [1], e.g. for applications such as AR/VR, online trading systems or even web surfing, comes a need to maintain short queues. Keeping queues short is inevitably hampered whenever packets are sent in bursts. This has rendered packet pacing—the insertion of time gaps in between packets—a corner-stone of many types of Internet communication today. Pacing, in various forms, has long become commonplace, e.g. in data centers, as a mechanism that is shipped in Linux, and in large-scale CDNs.

There are, however, significant difficulties in implementing pacing. Less than a decade ago, pacing was generally perceived as beneficial, but too hard to implement to be practical; this difficulty was typically attributed to the poor precision of software timers. Some hardware solutions were proposed [2], [3], [4], but they did not see mainstream deployment. Meanwhile, two opposing trends surfaced:

1) Software timers have become feasible (albeit not without problems, e.g. they are quite CPU-intensive), and software-based pacing has therefore become common. Linux, for example, supports it via its Hierarchical Token Bucket (HTB) and the FQ/pacing queuing discipline (see [5] for an excellent discussion of these and other implementations).
2) TCP Segmentation Offloading (TSO) and a more generic form of it (Generic Segmentation Offloading (GSO)) have become widely deployed. TSO allows the CPU to hand over a large block of data (usually 64 Kbyte) to the Network Interface Card (NIC), where it is dissected into packet-size data blocks. Packet headers are added to these blocks, and they are sent out into the network.

Because software timers do not have control over the individual packets that are created from such a larger data block, TSO is usually disabled when using pacing. As an alternative, Linux can dynamically change the size of the TSO data blocks[1]. Naturally, reducing the size of TSO blocks comes at a cost—e.g., the authors of [5] find that even using a software NIC to run their implementation improved its performance due to gains from larger TSO batching.

Can we get the best of both worlds—*can we offload the pacing operation together with segment offloading?* If this is possible, hw-based offloading can naturally operate on the packets that the NIC creates as the result of TSO. Such offloading requires three things to be efficiently and scalably implemented in hardware: (1) Buffer management and scheduling; (2) ACK parsing and timer calculation, as the inter-packet gap (IPG) of TCP packets is determined when ACKs arrive; and (3) quickly configurable and flexible timers.

In this paper, we make the case that such a comprehensive TSO+pacing operation is feasible. Indeed, related works (discussed in the next subsection) have assessed the feasibility of items 1 and 2 listed above. This paper specifically focuses on item 3, showing feasibility of an ultra-flexible and easily programmable HW timer. We believe that such initial step will give us the possibility to design and implement a comprehensive solution integrating all the three above aspects, as planned in our future work.

Specifically, this paper makes three contributions. First, we propose a hardware calendar and a relevant programming abstraction (Section II) which is at the same time very simple and flexible, as it permits programming packet-based pacing strategies that can be as fine-grained as permitting to change the IPGs between packets within a TSO data block. Our second contribution (Section III) is a methodology to translate a desired pacing pattern expressed in terms of a high-level rate envelope into a per-packet scheduling decision supported by our API. We illustrate the proposed approach also via examples taken from the TCP slow start and congestion avoidance stages. Finally, our third contribution is an FPGA implementation, described in Section IV, which relies on a *novel* insertion strategy for solving collisions in the calendar's hash table. Section V evaluates the novel insertion technique comparing it with a standard Cuckoo hash table. Section VI concludes the paper.

[1]See e.g. https://lwn.net/Articles/564979/

978-1-5386-5541-2/18/$31.00 ©2018 IEEE

A. Related work

We use SENIC [6] as the foundation for our claim that buffer management and scheduling can be efficiently implemented in hardware. Different from prior work such as vShaper [7], SENIC can directly support 10s of thousands of rate limiters, by dividing the work between the OS and the HW: the OS classifies packets in its host memory, and SENIC only needs to schedule and transmit packets by pulling them from this memory. However, SENIC does not support TSO, as it pulls MTU sized packet portions from the host memory, and there is also no particular support for quickly changing timers (SCENIC's timers are derived from a rate limit that is configured for transmit queues (classes) that the NIC exposes).

ACK parsing in HW is readily provided by either using programmable data plane technlogies/languages such as P4, or by using tailored packet-manipulation-oriented HW processors such as the one described in [8], and Section IV explains how we implemented our timer calculation; together, these two functions constitute the second piece of our puzzle.

There are various reasons to update NICs for low-latency communication beyond the "TCP vs. TSO" explanation that we have given in the previous section. Some reasons are provided in [9], and [10] describes how NICs can be extended to better isolate "bare-metal" data center guests (i.e., tenants who do not wish to run their software in a Virtual Machine).

Flexible timers are indeed necessary for pacing, as pacing decisions can depend on various factors. Initial Spreading [11], for example, aims at spreading the packets of TCP's Initial Window (IW) across an RTT to reduce the negative impact of a large initial burst. Trickle [12] is a mechanism specifically designed for YouTube, a very bursty application which poorly interacts with TCP. Reducing the size of the bursts that YouTube hands over may seem to be the natural solution, but doing so would incur increased overhead due to the larger number of write system calls as well as timers firing more often. Rather, Trickle rate-limits YouTube videos by artificially constraining the TCP sender's congestion window.

From the hardware point of view, the work most similar to ours is [13], where a flexible timer module was proposed. This module can be used to implement TCP offloading engines such as the one proposed in [14]. However, the implementation of this timer module uses a simple linear addressing in which there is a trade-off between the timer resolution and the number of timers to allocate (i.e., allocating more timers requires to decrease the timer accuracy). The achievable resolution reported in [13] is 1 ms. This is enough for timers used for triggering retransmissions, but cannot be used for pacing purposes. Generally speaking, hardware timer modules have been developed for TCP offloading engines [15], as well as for offloading other network functions [16]. All these modules have shortcomings regarding scalability (number of timers that can be allocated) or resolution granularity.

II. Calendar: architecture and API

A. Calendar Architecture

Figure 1 shows the high level (conceptual) architecture of the proposed programmable calendar. The underlying idea

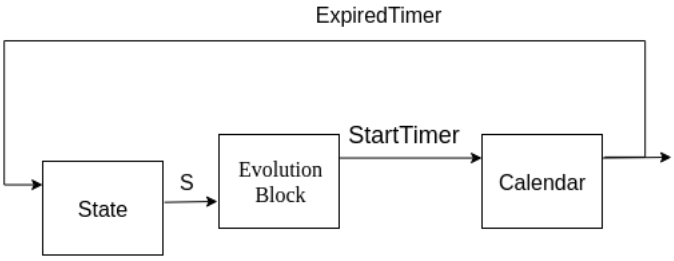

Fig. 1. Conceptual Architecture of the programmable calendar

is that every packet transmission (i.e., expired pacing timer) triggers the scheduling of the "next" transmission, further taking into account a system state. This permits to deploy a *stateful adaptive pacing strategy* where the time interval between two consecutive packets depends on a programmer-defined "state", which resides in the "State" block highlighted in Figure 1, and which may (at least in principle) be updated at every timer expiration. Specifically, being $\mathcal{S}$ a set of states, and $\mathcal{T}$ a time, the most general adaptive pacing function Δ supported by our approach would be a function

$$\Delta : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S} \times \mathcal{T}$$

which, given the transmission of packet n at time $T(n)$, and given a relevant system state $s(n) \in \mathcal{S}$, would return i) the time $T(n+1) > T(n)$ after which the next packet $n+1$ is scheduled for transmission, and ii) the next state $s(n+1)$. In practice, in the current HW implementation, we have so far resorted to the special case of state $s(n) \in \mathcal{S}$ being a counter, i.e. $s(n) = n$ and $s(n+1) = n+1$. Indeed, most use cases we have considered appear to be already accommodated by this choice. Note that the general implementation of the state block may leverage the HW developed in our past work [18].

The incremental computation of the next timer is performed by the "Evolution block" at each timer expiration, using as input the state of the system and the last expired timer. For each timer expiration, the block computes the next timer to insert into the Calendar. Generally speaking, we can consider the "Evolution block" able to provide any function $\delta(n) = T(n+1) - T(n)$; however, for sake of concreteness, in our implementation we have implemented the "Evolution block" as a simple look-up table. This design choice provides a high programmability level for the function that can be realized by the evolution block. It has as a drawback that the input resolution of the $\delta(n)$ function (the number of elements representing the function domain) is limited by the amount of memory allocated for the look-up table. Finally, the Calendar block receives the commands to insert timers from the "Evolution block" and signals that a timer is expired.

B. Application programming Interface

The Calendar is programmed by means of a basic Application Programming Interface (API) described hereafter using a notation close to the way a timer module is specified in [17]. The purpose of the calendar is to signal that a time interval has elapsed; usually a specific action is associated to the expiration of the timer. Since a calendar is able to manage a large number of timers, a specific ID is associated with each timer that is

inserted in the calendar. This ID allows binding the expiry of a timer to an action to execute. We can describe the calendar API with the following basic routines [17]:

1) *StartTimer(Interval, ID, ExpiryAction)*: The client calls this routine to insert a timer in the calendar that will expire after "Interval" units of time. The client provides a unique ID for each timer that is inserted in the calendar. When the timer expires, the ExpiryAction is executed.
2) *StopTimer(ID)*: The client calls this routine to remove the timer specified by the ID from the calendar.
3) *PerTickBookkeeping*: This routine checks which timers are expired in the current "Tick" interval. The "Tick" interval is the minimum time granularity of the calendar.

The first two routines are activated on client calls, while the last one is invoked on each timer tick. To measure the performance of the calendar we can refer to several specific figures of merit. One of the most important figures is the memory required to allocate timers. Since fast memory is a limited resource in hardware networking devices, it is important to optimize the memory allocated for each timer. Another figure of merit is the minimum time granularity that the calendar can support. Time granularity relates both to the *PerTickBookkeeping* routine and to the *StartTimer routine*. In the first case the time granularity corresponds to the minimum "Tick" interval that the calendar can support. In the case of the *StartTimer* routine it is the minimum Interval value that the Calendar can manage. As will be discussed in section IV, this is related to the worst-case insertion time of a new timer.

In fact, the setting of a timer with an Interval value t in a calendar is performed setting an expire time $T = t + t_0$, where t_0 is the current time. Unfortunately, the insertion of T is not immediate, but can require multiple clock cycles. We can define the worst insertion time of a timer in the caledar as t_{max}, so in the worst case the timer is actually inserted only at time $t+t_{max}$. Therefore a calendar is able to reliably insert a timer with Interval time t only if $t > t_{max}$. The hardware implementation that we developed (see Section IV) provides a time resolution well below 1 μs.

III. From Rate Envelopes to Timers

For simplicity of presentation, the analysis presented in this paper focuses on packet-based pacing. The extension to byte-based pacing, e.g. to further account for variable packet sizes if/when necessary, is relatively straightforward following the very same arguments reported below, and left to the reader.

A. Warm up example: exponential TCP slow start pacing

Before moving to the general case, it seems useful to illustrate via an initial simple example how our proposed timer-based pacing framework proposed can be easily related to a rate-based requirement emerging from an application. In this section, we focus on a classical use-case, namely *Exponentially Increasing Rate*, as encountered, e.g., in the TCP Slow Start rate growth pattern.

Let us first remind how TCP Slow Start (roughly) works. Let's index segments starting from 1, i.e., the first segment transmitted at time 0 is segment 1, and so on. Suppose that the transmission time T_{tx} for each TCP segment is negligible with respect to the Round Trip Time (RTT), and for convenience assume a constant and unitary $RTT = 1$ (i.e., time is measured in RTT and ACKs are not delayed). Under such assumptions, as long as no loss is encountered, the legacy TCP Slow Start procedure would yield the following transmission pattern:

- $t = 0 \rightarrow$ segment #1
- $t = 1 \rightarrow$ segment #2 followed by #3 (after $T_{tx} \approx 0$)
- $t = 2 \rightarrow$ segments #4, #5, #6 and #7 (separated by T_{tx})
- $\cdots$

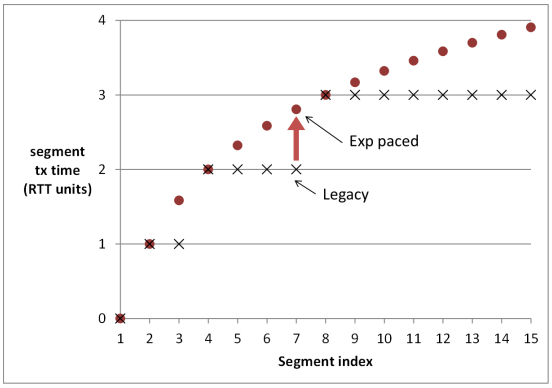

Fig. 2. Exponentially increasing rate pacing

What strikes in such classical Slow Start algorithm is that, due to the simple way in which the TCP protocol is implemented (congestion window increased by 1 at each ACK reception) a desired exponential growth in terms of transmission rate comes along with the undesired by-product of potentially severe burst transmissions[2]. Indeed, this pattern can be summarized by defining with $n(i)$ the *total* number of packets transmitted *before or "during"* the time instant i. Obviously,

$$n(i) = 2^i \tag{1}$$

with the size of a burst starting at each discrete time instant i thus being computed as $n(i+1) - n(i) = 2^{i+1} - 2^i = 2^i$.

If technically feasible, it would be quite natural to replace the Slow Start's traditional pattern with a *smoother* one. The obvious way to do so, as shown in figure 2, consists in spreading the transmission of segments such that equation (1) holds not only for discrete time instants i, but also for *any* intermediate real-valued time t, i.e. such that the number of transmitted segments up to time t should not be greater than 2^t. Being $n = 1, 2, \cdots$ the packet index, it readily follows that the time $T(n)$ at which packet n shall be transmitted is

$$T(n) = \log_2(n)$$

This rule would produce the very smooth pattern illustrated in figure 2. This pattern matches the slow start behavior expected in any discrete time multiple of an RTT (for instance, packet #8 is still transmitted, as mandated, at time 3), but spreads the transmission of packets in between—e.g., segment #7 would

[2]In practice, since two transmissions are triggered by each ACK reception, for relatively large ACK interarrival times, the pattern may not *exactly* converge into a single huge burst; still, in normal conditions, it certainly does not reduce to a smooth and regular pattern!

be paced and delayed until $T(7) = \log_2(7) = 2.807$, instead of being transmitted back-to-back right after packets #4-6.

Finally, it is worth to remark that, once we have an explicit expression for the time $T(n)$ at which packet n should be transmitted, we can readily determine an explicit *incremental* formula, such that the time elapsing before the "next" packet transmission is provided at the time at which the "previous" packet is transmitted (the transmitter needs only to keep track of the current packet index n). Indeed, if we define $\delta(n) = T(n+1) - T(n)$, for the above example case of slow start, it follows that

$$\delta(n) = \log_2(n+1) - \log_2(n) = \log_2\left(1 + \frac{1}{n}\right)$$

B. Extension to the general case

The above derivation is just a special case of a more general fluid model described hereafter. Let us consider the general case of a pacing strategy where the goal is to emit packets at an *instantaneous rate envelope*, formally expressed by an arbitrary function $R(t)$ [packets/time-unit].

If we interpret packets as continuous quantities, then the cumulative number $N(t)$ of emitted packets in a target time interval, say (t_0, t) can be any positive real-value quantity, which is related to the instantaneous rate $R(t)$ by the obvious integral equation

$$N(t) = N(t_0) + \int_{t_0}^{t} R(\tau)\mathrm{d}\tau \tag{2}$$

The above formula (2) holds for "fluidic" packets, i.e., it does not yet take into account the discrete nature of packet transmissions. To determine a desired pacing strategy we need to include in the above formula such a restriction, i.e. to impose that a new packet is transmitted at the instant in which the integral of the rate function $R(t)$ yields an integer value.

In practice, non restrictively assuming that $t_0 = 0$ and numbering packets starting from n_0, if the programmer wishes to enforce a given *rate envelope* $R(t)$, it suffices to solve, for any integer value n, the integral equation

$$n - n_0 = \int_{t_0}^{T(n)} R(\tau)\mathrm{d}\tau$$

in the set of unknowns $T(n)$, which will provide the transmission instant for the n^{th} packet.

In some cases (such as the TCP slow start example discussed before), rather than the rate envelope $R(t)$, the programmer may wish to directly start from a pacing requirement expressed in terms of a *cumulative packet transmission profile* $n_0 + N(t)$, i.e. the requirement that in a time window (0,t), no more than $n_0 + N(t)$ shall be transmitted. Owing to the relation (2), it follows that in such case the sequence $T(n)$ is directly obtained in closed form as the inverse function $T(n) = N^{-1}(n - n_0)$. The next use cases will provide some practical derivation examples.

Example 1: exponential pacing. It is instructive to revisit the example already discussed in section III-A, and show how it maps over the formalism introduced in this section. In that case the problem is simplified as the desired exponential packet emission profile is already expressed in terms of the cumulative number of transmitted packets. Hence, $N(t) = 2^t$ and therefore $T(n) = N^{-1}(n) = \log_2(n)$.

Example 2: linearly increasing pacing. The case of linear increase in the pacing rate is a bit more interesting for our purposes, because it is frequently encountered in real world scenarios (e.g., TCP congestion avoidance's additive increase phase), and because in this case (unlike the previous slow start one) the rate envelope computation $R(t)$ is not as trivial as the reader might expect.

Going into details, let us assume that the number of packets to be transmitted is increased by one unit per each time window, e.g. an RTT. Thus, if we start from a single packet just transmitted at time $t = 0$ (i.e., TCP's unitary congestion window), we will transmit one packet exactly after one RTT, two additional packets by the second RTT (for a total of three packets so far), three additional packets by the third RTT (hence a total of 6 packets), and, in general, $k(k+1)/2$ packets by the integer time instant k. Hence

$$N(t) = 1 + \frac{t(t+1)}{2} = 1 + \int_0^t R(\tau)\mathrm{d}\tau = 1 + \int_0^t \left(\tau + \frac{1}{2}\right)\mathrm{d}\tau \tag{3}$$

where $R(\tau) = \tau + 1/2$, the enforced rate envelope, can be intuitively justified by noting that — besides the linear increase τ — we also need to account for the fact that a packet transmission must occur at time $t = 1$ — the additional constant $1/2$ indeed follows from the "normalization" condition $\int_0^1 R(\tau)\mathrm{d}\tau = 1$.

Finally, by inverting (3) and computing it for integer values, we readily obtain the desired linearly increasing pacing rule in terms of instant of transmission $T(n)$ of the n^{th} packet as the solutions of the equation:

$$n = 1 + \int_0^{T(n)} \left(\tau + \frac{1}{2}\right)\mathrm{d}\tau = 1 + \frac{T(n)}{2} + \frac{T(n)^2}{2}$$

which, for any integer value $n > 0$, yields the positive solution

$$T(n) = \frac{\sqrt{8n-7} - 1}{2}$$

For the convenience of the reader, the first 11 values for $T(n)$ are here reported: $\{0, 1, 1.56, 2, 2.37, 2.70, 3, 3.27, 3.53, 3.77, 4\}$. As expected, starting from the transmission of packet #1 at time 0, packet #2 is transmitted at $t = 1$, packet #4 at $t = 2$, packet #7 at $t = 3$, and packet #11 at $t = 4$.

Finally, once the sequence of $T(n)$ values is given, an explicit *incremental* formula, which permits to schedule the "next" packet transmission by (only) keeping track of the current packet index n, is readily obtained as

$$\delta(n) = T(n+1) - T(n) = \frac{\sqrt{8n+1} - \sqrt{8n-7}}{2}$$

IV. FPGA IMPLEMENTATION

The programmable calendar has been implemented on a NetFPGA SUME as a component of the OpenState platform described in [18]. The table realizing the "Evolution Block" of Fig. 1 is a 4096x32 bits (16KB) memory block addressed with

the 12 least significant bits of the Interval values provided by the calendar block.

In order to realize a Calendar with a fine-grain time resolution and able to efficiently manage a large number of timers, we design an ad-hoc hardware module that fulfills all these requirements. We clocked the calendar with a 200MHz clock, corresponding to a 5 ns clock tick. The calendar uses a multiple-choice hash table [19] where the *expiration times* are stored. In particular, the *expiration time* represents the key to store in the hash table while *ID* and *ExpiryAction* are the values associated to the key. For the implementation we selected a 4x1 structure in which the hash structure is divided in 4 tables each one providing one bucket to store the {key,value} pair. The tables are realized using the dual port Block RAMs of the FPGA. Since the FPGA has several MB of memory available as dual port Block RAM, the calendar is able to store tens of thousands of timers. In our specific proof-of-concept we implemented the calendar using 32 Block RAMs (2% of the NetFPGA available Block RAMs), corresponding to 128 KB of memory, enabling to store more than 16 000 timers (8 Bytes for each timer).

The use of dual port RAMs allows using one port for the *PerTickBookkeeping* and the other for the *StartTimer* routine. The *PerTickBookkeeping* simply searches for the *present time* inside the multiple-choice hash table at each clock cycle performing an hash table lookup. Instead, the *StartTimer* routine is done as follows: (1) the *expiration time* is computed by adding the *present time* (with 5 ns resolution) to the Interval provided by the *StartTimer* routine; (2) the calendar tries to insert the *expiration time* in one of the tables. If there is a void slot the insertion is done; (3) if there are no void slots, the calendar continuously adds a clock tick to the *expiration time* and tries to insert the new value until a void slot is found.

We highlight that adding a clock tick to solve a collision occurring in a hash table is a specific (novel) approach that can be applied in our case since the shift of a few nanoseconds when a collision occurs gives a negligible impact on the calendar accuracy. Alternative approaches such as cuckoo tables [20] or de-amortized cuckoo hashing [21] are much more complex to implement and provide an insertion time that is worse than the one achievable by adding a clock tick.

In fact, the cuckoo algorithm first tries to insert the new key in a void slot (as in the calendar algorithm), and if no void slots are found randomly kicks out one of the keys already stored in the calendar and tries to insert this new key in a void location. So in the standard cuckoo implementation the first insertion would have 4 different slots to try, but all the subsequent tries would only have 3 slots to try (as one of the slot is already taken by the key that kicked out the key under insertion). Instead, the calendar algorithm always has 4 slots to try for each insertion. This intuitively explains why this insertion method provides an insertion time better than the standard cuckoo algorithm. In Section V a set of simulations are shown that confirm the intuition described here. The drawback of the calendar algorithm is that the actual expiry time is slighly different from the one indicated by the StartTimer routine. However, since the time resolution is extremely high, this small modification does not have any practical effect. Simulations presented in Section V show that the maximum error due to the insertion is 71 clock cycles (less than 400 ns) in the worst case of a fully loaded table.

Finally, it is worth noting that the standard collision resolution techniques are based on the hypothesis that multiple occurrences of keys with the same value cannot exist. This is not true in our case, where it is possible to have different timers with the same *expiration time*. This event can jeopardize the capability of the cuckoo algorithm to insert timers inside the tables. For example, if we try to schedule 5 timers to expire at the same time instant, the standard 4x1 cuckoo table fails to allocate these timers[3]. Adding a clock tick not only decreases the insertion time but also solves this "hard collision" event that however cannot be managed with standard techniques.

V. Results

To test the effects of the insertion algorithm we compared both the average insertion time and the worst case insertion time of a standard cuckoo hash with those achievable with our algorithm. As discussed in the previous section, the insertion time for the calendar insertion includes an error due to the difference between the expected expiration time set by the *StartTimer* request and the actual expiration time stored in the calendar. The calendar tries to insert the expiration time in one of the 4 locations of the hash table. Therefore, if we call l the hash table load factor, we can compute the probability $P(n)$ that a timer is inserted in exactly n tries as

$$P(n) = l^{4(n-1)} \cdot (1 - l^4) \tag{4}$$

and the average expected value can be computed as

$$E(n) = \sum n \cdot P(n) = \frac{1}{1 - l^4}. \tag{5}$$

To check the correctness of this model we simulated a hash structure composed of 4 hash tables of 4096 rows which first is loaded to a specific load value, and then a dynamic insertion/remove procedure is applied. The procedure substitutes each expired timer with a new one maintaining the same load factor for each insertion. This substitution has been applied 1000 times for each test and the test has been repeated 1000 times. This simulation corresponds to the worst case in which it is expected to have the worst insertion times.

The results of the simulation are shown in figure 3. The graph shows the average insertion time for different load factors achieved by simulation and the one computed using equation 2. The plot shows a good agreement between the simulated results and the computed ones.

The comparison with the insertion time of the standard cuckoo algorithm is reported in figure 4. For all the graphs the x-axis reports the load factor at which the hash table is loaded before starting the insertion and removal procedure. The plot in figure 4.a) reports the average insertion time for the standard and for the calendar insertion algorithms, while the plot in figure 4.b) reports the worst case insertion time

[3]Multiple insertions of identical keys are much more problematic than the extreme case provided here as an example. A more mathematical discussion of this effect can be found in [22].

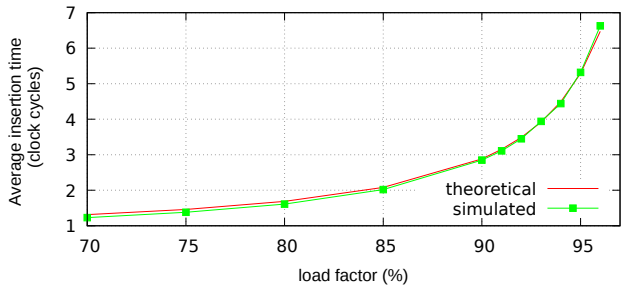

Fig. 3. comparison between simulated and theoretical average insertion time

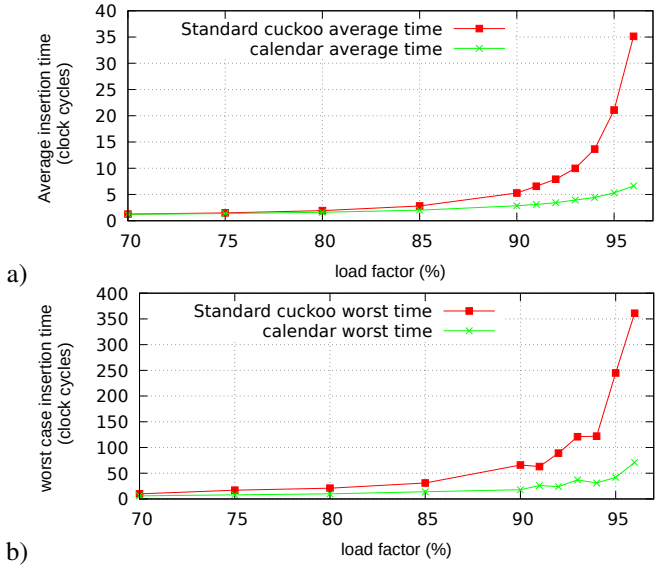

Fig. 4. Comparison between average insertion time of standard cuckoo and calendar. a) the plot reports the average insertion time. b) the plot the worst case insertion time.

for the two algorithms. As expected, the calendar insertion is always faster than the standard one, both on average and in the worst case. For this last parameter, it is worth to notice that the overall worst case, which occurs when the table is fully loaded (97%), is only of 71 clock cycles, corresponding to 355 ns. As expected, the time resolution of the calendar is much better than the target resolution of 1 μs.

VI. Conclusion

In this paper we have presented a flexible and programmable approach which permits to offload very accurate, fine-grained, and adaptive (stateful) packet-based pacing down into a hardware Network Interface Card. Our approach is as flexible as a Software timer, but since pacing is enforced in the NIC, it can be integrated with other important offloading techniques such as TCP Segmentation Offloading (TSO). Indeed, our work in progress consists in designing a comprehensive solution which, in addition to fully programmable HW pacing (this paper) further integrates the remaining primitives necessary to implement a full-fledget TSO-capable NIC, namely Buffer management and scheduling, and ACK parsing.

Acknowledgments

This work is partially supported by the European Commission in the frame of the Horizon 2020 project 5G-PICTURE (grant #762057).

References

[1] Briscoe, B., Brunstrom, A., Petlund, A., Hayes, D., Ros, D., Tsang, I.-J., Gjessing, S., Fairhurst, G., Griwodz, C., and Welzl, M., "Reducing internet latency: A survey of techniques and their merits", *Communications Surveys Tutorials, IEEE*, PP(99):1–1, 2014.

[2] Kobayashi, K., "Transmission timer approach for rate based pacing TCP with hardware support", in *Proc. of the Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2006.

[3] Hiraki, K. et al., "End-node transmission rate control kind to intermediate routers—towards 10Gbps era", in *Proc. of the Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2004.

[4] Takano, R. et al., "Design and Evaluation of Precise Software Pacing Mechanism for Fast Long-Distance Networks", in *Proc. of the Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2005.

[5] Saeed, A., Dukkipati, N., Valancius, V., Lam, V. T., Contavalli, C., and Vahdat, A.,"Carousel: Scalable Traffic Shaping at End Hosts". in *SIGCOMM '17*, pp. 404-417.

[6] Radhakrishnan, S., Geng, Y., Jeyakumar, V., Kabbani, A., Porter, G., and Vahdat, A., "SENIC: scalable NIC for end-host rate limiting", In *11th USENIX NSDI'14*, pp. 475-488.

[7] Kumar, G., Kandula, S., Bodik, P., Menache, I., "Virtualizing Traffic Shapers for Practical Resource Allocation", In *Proc. of the 5th Usenix Workshop on Hot Topics in Cloud Computing (HotCloud'13)*.

[8] Pontarelli, S., Bonola, M. and Bianchi, G.: "Smashing SDN 'built-in' actions: Programmable data plane packet manipulation in hardware", 2017 IEEE Conf. on Network Softwarization (NetSoft '17).

[9] Flajslik, M., and Rosenblum, M., "Network interface design for low latency request-response protocols", In *Proc. of the 2013 USENIX conf. on Annual Technical Conference (USENIX ATC'13)*.

[10] Mogul, J. C., Mudigonda, J., Santos, J. R., and Turner, Y., "The NIC is the hypervisor: bare-metal guests in IaaS clouds", In *Proc. of the 14th USENIX conference on Hot Topics in Operating Systems (HotOS'13)*.

[11] Sallantin, R., Baudoin, C., Chaput, E., Arnal, F., Dubois, E., and Beylot, A. L., "Initial spreading: A fast Start-Up TCP mechanism", In *Proceedings of the 38th Annual IEEE Conference on Local Computer Networks (LCN 2013)*, pp. 492-499.

[12] Ghobadi, M., Cheng, Y., Jain, A., and Mathis, M., "Trickle: rate limiting YouTube video streaming". In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 17-17.

[13] C. Neely, G. Brebner, and W. Shang, "Flexible and modular support for timing functions in high performance networking acceleration," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on.* IEEE, 2010, pp. 513518.

[14] Sidler, D., Alonso, G., Blott, M., Karras, K., Vissers, K., and Carley, R. "Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware," In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on* (pp. 36-43). IEEE.

[15] Ding, L., Kang, P., Yin, W. and Wang, L., "Hardware TCP Offload Engine based on 10-Gbps Ethernet for low-latency network communication," In *Field-Programmable Technology (FPT), 2016 IEEE International Conference on* pp. 269-272.

[16] Antichi, G., Shahbaz, M., Geng, Y., Zilberman, N., Covington, A., Bruyere, M., McKeown, N., Feamster, N., Felderman, B., Blott, M. and Moore, A.W., "OSNT: Open source network tester," *IEEE Network, 28(5)*, pp.6-12.

[17] Varghese, G. *Network algorithmics.* Chapman & Hall/CRC, 2010.

[18] Pontarelli, S., Bonola, M., Bianchi, G., Capone, A., and Cascone, C., "Stateful Openflow: Hardware Proof of Concept," In *IEEE High Performance Switching and Routing (HPSR)* (2015).

[19] Mitzenmacher, M. "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems* vol. 12, no. 10 (2001), pp. 1094–1104.

[20] Pagh, R., and Rodler, F. F. "Cuckoo hashing," *Journal of Algorithms* vol. 51, no. 2 (2004), pp. 122–144.

[21] Kirsch, A., and Mitzenmacher, M. "Using a queue to de-amortize cuckoo hashing in hardware," In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing* (2007), vol. 75.

[22] Lelarge M. "A new approach to the orientation of random hypergraphs," In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, 2012, pp. 251-264.