Measuring and Reducing the Impact of the OS Kernel on End-to-End Latencies in Synchronous Packet Switched Networks

# Measuring and Reducing the Impact of the OS Kernel on End-to-End Latencies in Synchronous Packet Switched Networks

Michele Welponer*, Luca Abeni*, Guido Marchetto† and Renato Lo Cigno*
* DISI, University of Trento, Via di Sommarive 14, Trento (Italy)
† DAUIN – Politecnico di Torino, 10129 Torino, Italy

Author's version
Published in
*Software: Practice and Experience*, vol. 42 n. 11, pp. 1315-1330
The final published version is accessible from here:
http://dx.doi.org/10.1002/spe.1134


**Abstract**

This paper presents an evaluation of the impact of the so-called OS latencies on the performance of a synchronous network based on global time coordination. The concept of end-to-end latency is first defined, by extending the concept of latency used to evaluate the performance of real-time systems, and the end-to-end latency provided by a general-purpose OS is measured as a benchmark. Finally, Real-Time techniques are used to reduce the worst case values of such a latency, showing how a gateway between synchronous and asynchronous networks can be implemented by using commercial-off-the-shelf hardware and a proper software stack (based on a Real-Time version of Linux). The use of a Real-Time OS is still a non-trivial task, which requires experience and the analysis of the specific application to devise the proper techniques to be applied. This work dissects the problem of OS-to-Network data transfer (and vice-versa) identifying the key sources of latencies and delay jitter, and solving each problem with the application of a proper technique.

**Keywords**

*Kernel Latency, End-to-end delay, Pipeline Forwarding, Real Time Linux*

## I. Introduction

Some of the dominant Internet paradigms have been recently challenged in light of performance bottlenecks and energy consumption issues [13], [14], [18], [19]. The utilization of network synchronisation to drive packet scheduling and forwarding can be a viable solution to alleviate both these problems. For instance, Time Driven Switching (TDS) has been proposed as a means to boost performance and, at the same time, reduce energy consumption of the Internet [4], [23]. Similar proposals include Time Driven Priority (TDP) [21], TWIN [26], and some Optical Burst Switching (OBS) based systems [24], [32], [35]. With the main purpose of guaranteeing deterministic latency delivery, this operating mode has been also proposed for wireless multi-hop networks (e.g., DARE [10], TAF [5]) and in the industrial scenario (e.g., Synchronous TDMA [31], [33] or Synchronous Ethernet (IEEE Standard 1588v2)).

At the heart of increased performance and reduced consumption of synchronised approaches, it lies the idea of Pipeline Forwarding (PF), which in practice means that information is forwarded "on the flight" between the source and the destination, without the need for intermediate buffering to process or store the information. As it will be shown in this paper, PF implies some strict real-time requirements, which are difficult to satisfy when the pipeline is implemented in software.

This project has been partially funded by the Italian MIUR PRIN Project BESOS ( Bandwidth efficiency and Energy Saving by sub-lambda Optical Switching) - Year 2008, prot. 2008P2HWBK_003

Pipelining is a well known technique used in many different fields, from production to transportation, but to work properly it requires a strict synchronisation between all the pipeline stages. In distributed systems this means that a *global coordination* is required. When the concept is applied to packet switching and routing, i.e., in PF, this means that forwarding packets with no buffering is only possible if packets are sent and received according to an appropriate schedule and such a schedule is strictly respected. For example, if a router receives at the same time two packets sent to the same interface, it is not able to properly forward them without buffering. Such a strict synchronisation between routers and end systems can be achieved by means of global time references such as Universal Time Coordinated (UTC), or Global Positioning System Time (GPST), the time used by the global navigation system, which is available also for civilian use for free and without restrictions in normal times. Future navigation systems like the European Galileo and the Chinese BeiDou will only improve the availability of a global time reference. Alternatively, several solutions for distributing the synchronisation reference directly through the network can be adopted; see for instance [2], [3], and, specifically for TDP, [7].

Although the availability of global time empowers global coordination, it does not mean that synchronisation needed for PF comes for free, as it requires the ability, for both end systems and routers, to send packets "at the correct time" to properly respect the schedule. Specially considering the end systems this may be a challenge because of the so called "latency", which can be informally defined as the difference between the time when a packet should be received(sent), and the time when it is actually received(send). While modern hardware (even off-the-shelf hardware) is generally able to provide the low latencies needed for PF, the software stack is more problematic, as the latencies generated by an Operating System (OS) kernel can be large enough to compromise the correct forwarding of packets or to force to use a low bitrate, as it will be shown in Section III.

While some works related to high-performance network applications only consider average throughput and do not care about the latencies introduced by the OS [8], [28], it has been recently noticed that controlling the maximum latencies can be important too [11]. For a synchronised service, average latencies are not a key issue, since the exact synchronisation point can be set to compensate this average, while the jitter and *worst case latencies* are the key performance metrics: the hard-real-time nature of global coordination means that compensation is not feasible, thus all latency variations will end up in overhead (guard-bands in the time frame), or in forwarding errors.

Such latencies have been defined, measured, and characterised in some previous works [1], [25], which however focused on the latencies in a single system, not connected to a network. In order to consider the effects of latencies on PF, some kind of *end-to-end* latencies have to be defined and measured. Hence, this paper extends to distributed systems the previous definitions of latency, performing an analysis of the performance of a software system implementing PF. A first version of such a system (based on a modification of a non-real-time operating system) is evaluated first, and real-time techniques [9], [27] are then used to reduce the impact of the software stack on both the worst case end-to-end latency and the resulting jitter. Since TDP can be easily combined with conventional IP routing (while still guaranteeing deterministic quality of service and low complexity packet scheduling), it has been selected as a test-case in this work. In particular, the analysis of the OS kernel contributions to the end-to-end latencies has been performed considering a software TDP implementation [7].

TDP is designed to serve time-sensitive traffic (such as the traffic used in real-time multimedia services) without affecting or changing the existing internetworking "best effort" data services. In particular, packets scheduled for transmission during a timeframe are given the maximum priority; if resources have been properly reserved, all scheduled packets will be at the output port and transmitted before the timeframe ends.

The results presented in this paper show that the worst case end-to-end latency can be reduced by a significant amount (when considering two TDP routers connected by a single link, it can be reduced to about $1/10$th). Furthermore, splitting this latency in its access and receive components can lead to separate compensation and thus to further reduce the overhead required to run the system.

The remaining part of the paper is organised as follows: Section II introduces the principles of globally synchronised networking; Section III defines the problem addressed in the paper and describes the end-to-end latency and its components; Section IV presents the real-time tests and optimisation which are the core of the contribution, and finally Section V ends the paper with a short discussion of the achievements and a roadmap for future development.

## II. Underlying Principles and Technologies

As the context of this work is a network performing PF of packets, this section briefly introduces this technology and its deployment options. An extensive and detailed description of pipeline forwarding is outside the scope of this paper and is available in the literature [2], [6], [7], [21], [23].

In PF all packet switches utilise a basic time period called *time-frame*. The time-frame duration $TF$ may be derived, for example, as a fraction of the UTC second received from a time distribution system such as the GPS. As shown in Figure 1, time-frames are grouped into time-cycles. This timing structure aligned in all nodes constitutes a Common Time Reference (CTR).

During a resource reservation phase time-frames are partially or totally reserved for each flow on the links of its route. Thus, time-frames can be viewed as virtual containers for multiple packets that are switched and forwarded according to the CTR. In the PF deployment in the literature, the time-cycle provides the basis for a periodic repetition of the reservation. In another

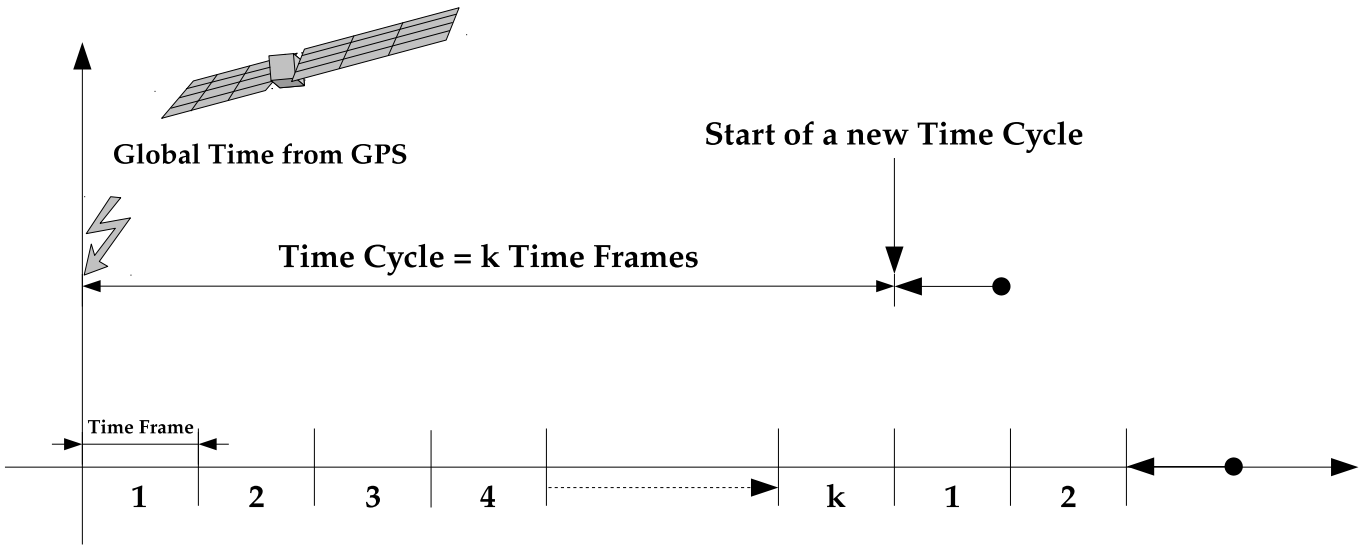


Fig. 1. GPST framing organised in periodic cycles of $k$ time-frames.

possible deployment the reservation phase can be done on the fly before transmitting a packet without necessarily maintaining it across multiple time-cycles.

A signalling protocol must be chosen for performing resource reservation and time-frame scheduling, i.e., selecting the time-frame in which packets belonging to a given flow should be forwarded by each router. Existing standard protocols and formats should be used whenever possible. Many solutions have been proposed for distributed scheduling in pipeline forwarding networks [30] and the generalised MPLS (G-MPLS) [15] control plane provides signalling protocols suitable for their implementation. Traditionally, if QoS is supported, applications signal their requirements to the network for each individual flow. The network in turn will reserve resources to the flow, inspecting each incoming packet to assign it to the correct flow. This is the model of the Integrated Service, using RSVP for signaling, but also of older models like ATM User-Network Interface. These models do not scale due to the large status information that has to be maintained in intermediate nodes, and also due to the complexity of per-packet classification required. To solve this issue the Differentiated Service model, which does not require complex per-flow classification, but only coarse packet classification based on a class, has been proposed, but the QoS is only stochastically guaranteed to individual flows. Pipeline forwarding does not require to maintain the status of each single flow, nor it does require to perform complex per-packet classification in intermediate nodes, thus it has similar provisioning scalability as the Differentiated Service model, where micro-flows are aggregated in the network to improve scalability, and QoS guarantees similar (indeed better, due to an intrinsic smaller forwarding latency and jitter) to the Integrated Service model. In essence, each TF is assigned to one macro-flow. During that TF, packets belonging to the macro-flow acquire highest priority.

The basic pipeline forwarding operation is regulated by two simple rules: (*i*) all packets that must be sent in time-frame $i$ by a node must be in its output ports' buffers at the end of time-frame $i-1$, and (*ii*) a packet $p$ transmitted in time-frame $i$ by a node $N_n$ must be transmitted in time-frame $i+\alpha$ by the following node $N_{n+1}$, where $\alpha$ is a predefined integer, and time-frame $i$ and time-frame $i+\alpha$ are also referred to as the forwarding time-frame of packet $p$ at node $N_n$ and node $N_{n+1}$, respectively. It follows that packets are timely moved along their path and served at well defined instants at each node. Nodes therefore operate as they were part of a pipeline, from which the technology's name is derived. Consequently, given the time-frame at which a packet enters the network, the time at which the packet is forwarded by each node and eventually reaches its destination is known in advance with the accuracy of one time-frame. In essence, PF guarantees that reserved real-time traffic experiences bounded end-to-end delay, low delay jitter independent of the number of nodes traversed, and neither congestion nor losses due to buffer overflow. Non-reserved traffic is instead transmitted in unused TFs, thus obtaining the traditional best-effort service.

The value of $\alpha$ is determined at resource-reservation time and must be large enough to satisfy rule (*i*). Note that the time a packet requires to go from the output buffer of a node to the output buffer of the following one is strictly dependent on the performance of both nodes and the distance between them. Thus, the minimum value acceptable for $\alpha$ depends on the previous hop from which a packet is received.

Among the possible deployment options, *Time Driven Priority* (TDP) is a synchronous packet scheduling technique that

combines PF with conventional routing mechanisms to achieve high flexibility together with guaranteed service. While scheduling of packet transmission is driven by time, the output port can be selected according to either conventional IP destination-address-based routing, or multi-protocol label switching (MPLS), or any other technology of choice. Within a time-frame packets can be switched and forwarded asynchronously, i.e., in an arbitrary order and to different output ports. Other PF-based technologies exist that are more suitable for high-speed (optical) networks. For example, in *Time-driven switching* (TDS) all packets in the same time-frame are switched together, i.e., to the same output port. Consequently, header processing is not required, which results in low complexity (hence high scalability) and enables optical implementation.

## III. Analysis of the Problem

As outlined in the previous section, modern synchronous communications based on PF are characterised by *global* coordination based on time frames[1]. For example, Figure 1 shows the timing organisation of a stage of the pipeline (a network node), which sends packets according to a cyclic schedule, defined by the repetition of $k$ *time-frames* of size $TF$. Each period of $k$ time-frames is a time-cycle.

The cyclic schedule is described by a table (the *scheduling table*, composed by $k$ rows) in which each row corresponds to a time-frame, and indicates how many packets of which flow should be sent in that time-frame.

The overhead in such systems comes from the need for global coordination, which is obtained through proper signals from a global coordination system. A dedicated module periodically triggers interrupts to define time-frame boundaries; however, the global coordination may be jeopardized by latencies introduced by sub-systems that directly manage these interrupts. To focus the point and without loss of generality, in this work global coordination is assumed to be achieved via GPST.

A GPS card is used as an interrupt source, and it is programmed to generate a periodic interrupt request (referred to as *GPS IRQ* from now on) at the beginning of every time-frame (hence, interrupts are generated with period $TF$). When a GPS IRQ fires, the sub-system must immediately look at the proper row of the scheduling table and send out all the packets that have to be sent in the current time-frame according to the table. If the time-driven communication protocol is entirely implemented in hardware (for example, in a TDS switch), the latencies between a GPS IRQ and the time when packets are actually sent can be reduced to a negligible amount. However, when considering implementations based on commercial off-the-shelf hardware plus dedicated software, such latencies can be quite large, especially if the Operating System kernel (which is responsible for several operations such as managing multiple tasks, servicing interrupts, and handling physical devices) has not been designed for respecting the tight temporal constraints required by PF. An example of such peripheral systems is the TDP Router presented in [6], [7].

### A. Using a Non Real-Time OS

Authors of [7] implemented a TDP router based on a non-real-time operating system (FreeBSD[2]). When a GPS IRQ fires, the router must immediately send out all the packets that have been scheduled for that time-frame. Hence, the TDP router works as follows:

- The TDP router is programmed to follow a *time-driven schedule*;
- The time-driven schedule is described by a scheduling function $S(i) = (f, n)$, with $f$ indicating the flow to be sent in the $i^{th}$ time-frame, and $n$ indicating the number of packets of the flow $f$ to be sent in such a time-frame. The schedule is periodic and the function $S(i)$ needs to be defined only for $0 \leq i < k$ (in other words, $S(i)$ can be coded in the scheduling table with dimension $k$).
- When a GPS IRQ fires, a new time-frame begins. The *time-frame counter* `tfc` is increased (`tfc = (tfc + 1) % k`), and the router reads the $\texttt{tfc}^{th}$ entry in the scheduling table: if $S(\texttt{tfc}) = (f, n)$, then the router immediately sends $n$ packets from the $f^{th}$ packet queue.

In order for the TDP router to work correctly, it is necessary to send the packets as soon as the GPS IRQ fires, minimising the worst-case latency between the interrupt and the actual transmission, otherwise the packets risk to be sent out in the wrong time-frame. As a consequence, a proper amount of time must be allocated in every time-frame to compensate for the uncertain latency in transmission.

To reduce such a latency, some design choices concerning interrupt priority levels have been done. In traditional OS kernels, interrupt requests are generally handled by short Interrupt Service Routines (ISRs) that acknowledge the hardware interrupt and activate a longer *bottom half* (also called *software interrupt*) which will be in charge of really processing the interrupt. FreeBSD offers the opportunity to register very high priority interrupts (defined as type `MISC`), which can preempt any other execution unless the interrupt service routine of the system clock. However, this interrupt type is preemptable by any other incoming interrupt, thus requiring proper protection of the associated interrupt service routine. It is also worth noticing that packet dequeue

[1]Traditional synchronous, isochronous, and plesiochronous communications, like SDH (Synchronous Digital Hierarchy), are instead based on local coordination with phase locking and local framing between transmitter-receiver pairs, which implies bit or byte stuffing to compensate for the lack of coordination between successive segments of a communication path.

[2]http://www.freebsd.org

operations performed within the GPS IRQ service routine of a TDP router are very short as packets are simply marked as dequeued and then independently moved to the NIC through DMA transfers. This considered, the authors of [7] registered the GPS card interrupt as a `MISC` interrupt, they protected its service routine against preemption by blocking all interrupts, and they implemented dequeue operations directly in the ISR, without triggering any bottom half. No modifications have been done to the type and priority of networking interrupts: ISRs are fired by network cards, but the vast majority of the networking code runs in bottom halves.

These design choices allowed the TDP router to offer reasonably low transmission latencies, but further improvements are required to adopt the router in high-performance optical networks. This is shown in the next subsection and motivates this work.

For example, some initial experiments, based on a qualitative analysis, indicated that the time-frame size $TF$ must be set to at least $250\,\mu$s in order to obtain a reasonably low packet loss ratio. To better understand and quantify the causes of such lost packets, some more quantitative and accurate measurements have been performed.

### *B. Experimental Setup*

To evaluate the performance of a TDP router two routers have been connected through an optical Ethernet. They will be referred to as the *sender* and the *receiver* in the following. The two TDP routers are based on the same hardware: a DELL server with two Intel Xeon64 CPUs running at 3.6 GHz and 1 GB of RAM. A Symmetricom bc635PCI-U GPS card is installed on both the two computers and it is used as the GPS IRQ source. The GPS cards provide GPST with an accuracy better than $1\mu$s[3]. The hardware has been configured to improve the predictability (for example, the Hyper-Threading feature of the CPU has been disabled through the BIOS, because such feature can increase the average CPU throughput but introduces unpredictable delays).

The sender has been configured to send a UDP packet to the receiver at the beginning of each timeframe. Such UDP packets have an application level payload of 36 bytes, resulting in a frame of exactly 102 bytes including Ethernet header, trailer, preamble, and inter frame gap (equivalent to 12 bytes) with a transmission delay $\simeq 0.816\,\mu$s. The propagation delay is negligible ($< 0.1\,\mu$s). In these first experiments, the TDP routers were not stressed with any additional load. The performance of the TDP router was measured by computing the difference between the GPS time when a packet was received by the receiver and the GPS time when the packet was supposed to be transmitted by the sender, which is the nominal beginning of the time-frame.

The actual measure is performed as follows. As soon as packets are passed from the network card to the operating system, the receiving device queries the GPS card and stamps the event with the measured GPS time $t_2$. The latency can be measured by computing the difference between $t_2$ and the beginning of the time-frame. Unfortunately, also the receiving device is affected by latencies deriving from the operating system operation. Thus, the measured value includes the unpredictable latencies experienced at both the sender and the receiver, plus the constant transmission and propagation delays, as explained in Section III-D.

### *C. Unpredictabilities in the Latency*

The setup described in the previous subsection has been used to measure the End-to-End latency experienced by the FreeBSD-based implementation when using different time-frame sizes ranging from $TF = 100\,\mu$s to $TF = 500\,\mu$s. The Probability Mass Functions (PMFs) of the measured End-to-End latencies over more than 10 hours have been measured. The results are reported in Figure 2, and they clearly indicate that the use of short time-frames ($TF < 200\,\mu$s) with a non real-time operating system kernel is not feasible. In fact the weird PMF in the $TF = 100\,\mu s$ plot are due to the fact that in practice the experiment measures the latency by computing the difference between the time when a packet is received and the beginning of the current time frame. Hence, if $L$ is the latency, the experiment really measures $L\%TF$ (and not $L$), thus packets arriving in the next time-frame wrap around in the measure. For $TF = 200\mu s$, the measurement showed a worst-case latency of $114\mu s < TF$. However, some additional experiments showed some lost packets. This is probably due to a worst-case end-to-end latency larger than $200\mu s$ which was incorrectly measured as $L\%TF$. For $TF = 250\,\mu$s and $TF = 500\,\mu$s the measure indicates that packets arrive within the intended time-frame, and additional experiments did not highlighted any packet loss. Still, for $TF = 250\mu s$ the measured latency is very large and implies a large overhead (notice the presence of a few packets with latency larger than $240\,\mu$s). Table I reports the worst-case latencies measured for the various time-frame sizes $TF$.

Even in absence of additonal workload, the end-to-end latency experienced by a TDP router implemented over a non real-time OS kernel is too high for using time-frame sizes $TF < 250\mu$s, and even with such large time-frames, the overhead is quite high.

### *D. Formal Definition and Analysis of Latency*

This section presents a more systematic study of the problems highlighted in the previous experiments, based on more accurate definitions.

As already mentioned, the lost packets problem can be explained by noting that the TDP router is supposed to forward packets immediately at the beginning of each time-frame, but in practice this is not the case. As a matter of fact, the packets are delayed by a random delay $L^{Tx}$. This delay is caused by the task management and the scheduling algorithm of the operating

[3]http://www.symmetricom.com/media/files/downloads/product-datasheets/DS_bc635PCI-U.pdf

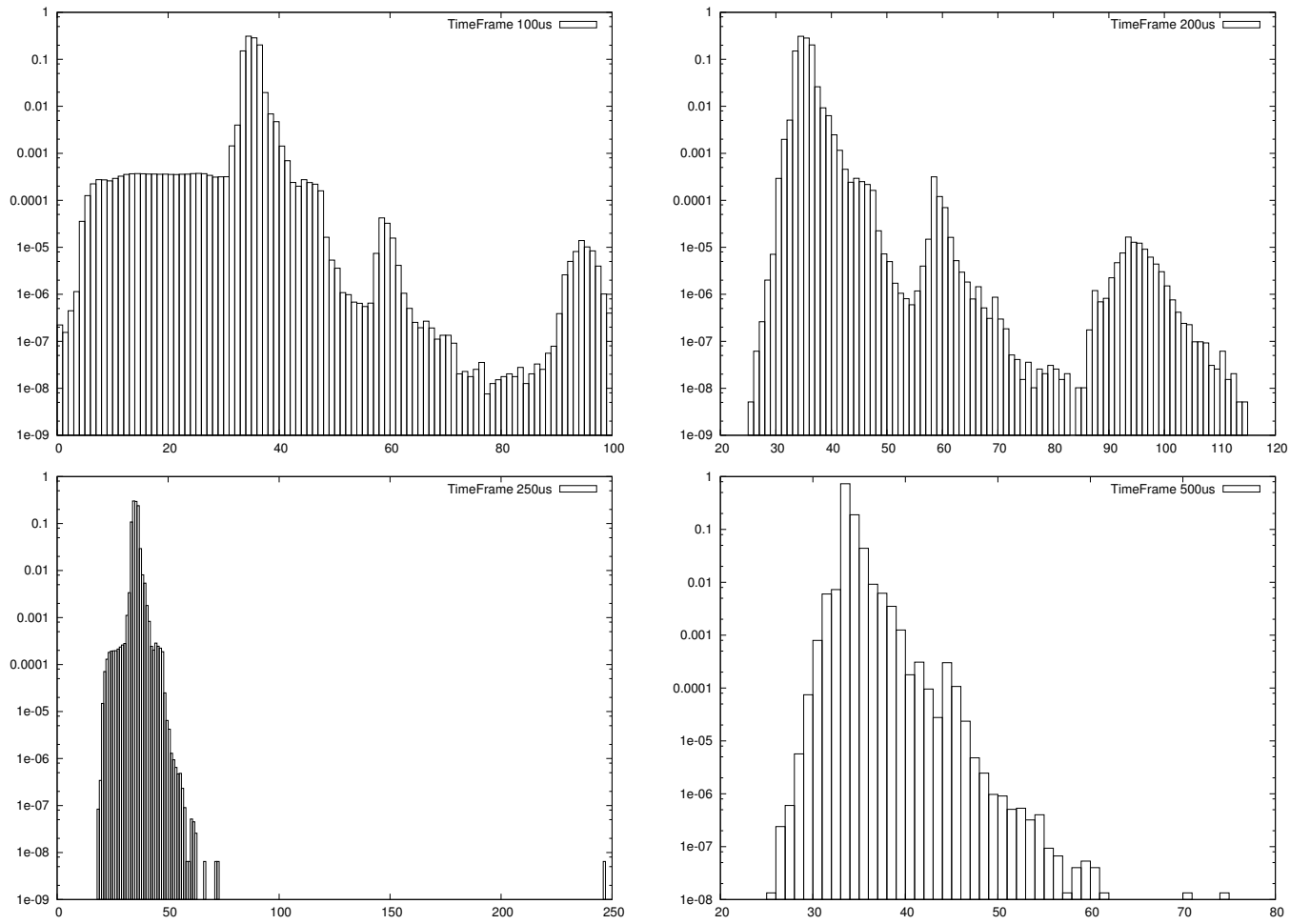


Fig. 2. Probability Distribution Functions of the end-to-end latencies for different sizes of the time-frame.

TABLE I. WORST-CASE END-TO-END LATENCIES.

| Time Frame size $TF$ | Worst Case Latency | Average Latency |
|---|---|---|
| $100\mu s$ | $> 100\mu s$ | $34.537\mu s$ |
| $200\mu s$ | $114\mu s$ (values $> 200\mu s$ probably exist) | $34.726\mu s$ |
| $250\mu s$ | $246\mu s$ | $34.821\mu s$ |
| $400\mu s$ | $101\mu s$ | $33.606\mu s$ |
| $500\mu s$ | $74\mu s$ | $33.342\mu s$ |

system kernel [1], and can lead to situations in which the time needed to send all the scheduled packets exceeds the size of the time-frame.

For a synchronised communication system to work properly, it is important to ensure that an upper bound for $L^{Tx}$ exists and is known (note that such an upper bound defines the overhead of the synchronised communication system). The average delay $\overline{L^{Tx}}$ is instead less important, and can also be partially compensated if there exists a minimum delay $L^{Tx}_{\min}$ that can be offset at the GPS IRQ level. $L^{Tx}$ can be better formalised as follows.

**Definition** The *Transmission latency* is defined as the delay $L^{Tx} = t_1 - t_0$, where $t_0$ is the time when the GPS IRQ fires (a time-frame begins) and $t_1$ is the time when the first packet sent in such time-frame is actually transmitted on the medium.

Unfortunately $L^{Tx}$ is very difficult to measure. First of all, because some media — such as the optical fibers, which are the media of choice for very high speed networking — do not allow easily to spill signals for measurements. Second, because

an accurate evaluation of $L^{Tx}$ requires perfect synchronisation of the measurement device with the GPS IRQ internal to the transmitter. For this reason, in this paper the whole End-to-End latency will be measured, instead of trying to measure $L^{Tx}$ in isolation.

**Definition** The *End-to-End latency* is defined as the delay $L = t_2 - t_0$ between the GPS IRQ firing time $t_0$ and the time $t_2$ when the first packet sent in the time-frame is delivered to the operating system of the remote host. By definition

$$L = L^{Tx} + \delta + L^{Rx} \tag{1}$$

where $L^{Tx}$ is the Transmission latency, $L^{Rx}$ is the unpredictable latency introduced by the operating system kernel on the receiving side and $\delta$ is the (nearly constant) propagation and transmission delay.

The propagation delay component of $\delta$ on standard optical fibers is 5 ns/m and is thus entirely negligible (far less than a $\mu$s) for direct connections with short fibers as in our testbed. Assuming a full Ethernet frame with a payload of 1500 bytes, the transmission delay component on a Gbit Ethernet link is $\simeq 12.6\,\mu$s, and reduces to $\simeq 0.75\,\mu$s for a minimum size frame with 46 bytes payload. In any case $\delta$ can be easily computed and it is deterministic for all practical purposes, thus it does not represent a measurement impairment.

To use small $TF$ values, it is necessary to reduce the end-to-end latency $L$, and in particular its component due to the OS kernel on the sender, which is the transmission latency $L^{tx}$. The receiving latency per-se is not important (packets will enter again the non-synchronised realm); however, in case of further forwarding to synchronised subsystems minimising also the receiving latency can only bring benefits. First of all, note that this latency reduction can be achieved only after understanding the sources of the various latencies.

The concept of latency is not new in computer science, and the latencies generated by an OS kernel have already been studied, measured, and reduced in single-computer systems [1]. This work is going to extend to distributed systems the previous measurements and latency reductions. In particular, the real-time performance of a single computer are affected by the so called *kernel latency*, while in distributed systems the end-to-end latency should be considered instead.

The transmission latency $L^{Tx}$ can be split in two components $L^I$ and $L_b^{Tx}$. The former is mainly affected by the time needed to react to a hardware interrupt and hence is not related to packet transmission; the latter is instead related to the transmission of packets at the TDP node. Notice that the kernel latency considered in previous papers is $L^I$, which in this paper will be referred to as *interrupt latency* due to its nature.

**Definition** The *Interrupt Latency* (or *Kernel Latency*) is defined as the time interval $L^I = t' - t$ between the time $t$ when a hardware interrupt is supposed to fire, and the time $t'$ when the OS kernel reacts to such an interrupt request.

The end-to-end latency defined in Equation 1 can be split in more components to highlight the effects of the interrupt latency; in particular, the transmission latency can be written as $L^{Tx} = L^I + L_b^{Tx}$, where $L_b^{Tx}$ is the latency component due to the data transfer to the NIC (the delay $t'' - t'$ between the time $t'$ when the OS kernel tries to send a packet and the time $t''$ when the packet is actually sent by the NIC). Hence, $L = L^I + L_b^{Tx} + \delta + L^{Rx}$ (in theory, the interrupt latency is a component of the latency both in transmitting and receiving packets, so $L^{Rx}$ could also be split in an interrupt latency plus a second component, but measuring the two components separately requires modification of the NIC. Hence, such a split is not considered here).

Since, as already pointed out, a direct measurement of $L_b^{Tx}$ and $L^{Rx}$ is not simple, in this paper the interrupt latency $L^I$ will be measured and reduced first. Then, the whole end-to-end latency (also including $L_b^{Tx}$, the receiving latency $L^{Rx}$, the propagation and the transmission delay $\delta$) will be measured (and reduced).

The interrupt latency $L^I$ can be reduced by using a proper Real-Time OS (RTOS) kernel and some specific real-time techniques and by using a dual-CPU system (which allows to use the so called *CPU shielding technique*). Finally $L_b^{Tx}$ and $L^{Rx}$ can be reduced by properly configuring the network cards and their drivers (note that the CPU shielding technique mentioned above is also important to reduce $L_b^{Tx}$ and $L^{Rx}$).

$L^I$ can be easily measured by periodically generating an interrupt, and by measuring the time when the interrupt handler is executed. The difference between such a time and the interrupt generation time (which is known in advance if a hardware timer is used to generate the interrupt) is the interrupt latency. Of course, the worst case latency can depend on the interrupt source: for example, it can be expected that the latencies experienced by interrupts generated by devices which are directly connected to the CPU are smaller than the latencies experienced by interrupts generated by devices which are connected to the CPU through a bus, normally the PCI bus, and are affected by the bus controller. Hence, $L^I$ has been first measured by using the CPU APIC timer as an interrupt source, and then by using the GPS card as an interrupt source (this second latency will be referred to as *GPS IRQ latency*, from now on).

The interrupt latency experienced when using the APIC timer as an interrupt source can be measured by using the standard `cyclictest` utility[4], and can provide a lower bound for the GPS IRQ latency. Thus, this measurement is important to understand if the hardware and the OS kernel can potentially provide the real-time performance needed for TDP operations.

[4]https://rt.wiki.kernel.org/index.php/Cyclictest

On the other hand, the GPS IRQ latency must be considered in case of PF, because the packets must be transmitted on the network interface based on a global timing coordination (GPS) rather than on the local clock of the machine. The GPS IRQ latency is greater than the APIC IRQ latency mainly because the interrupt is issued through the PCI bus. Moreover its measure is also more complex, since it required to use the same GPS time and not the internal CPU clock. The GPS IRQ latency is measured as follows:

1) the GPS card is programmed to generate one interrupts exactly at the beginning of each time-frame;
2) when the interrupt handler is executed, UTC $t'$ is read on the GPS card;
3) the latency is estimated as $\hat{L}^I = t' \,\%\, \text{TF}$, because it is known that the interrupt was supposed to be generated at time $t = k\text{TF}$, with $k = 1, 2, ....$

If $L^I < TF$ the above estimate is correct but for the unknown, random time access to the GPS card, which is however limited to a few bus cycles (tens of ns), thus negligible compared to the interrupt latency.

## IV. Reducing the Worst-Case End-To-End Latency

This section describes how the end-to-end latency defined and measured in Section III can be reduced by using real-time techniques, so that a TDP router can be implemented without too much overhead. Real-time literature provides many techniques and solutions (ranging from advanced scheduling or resource allocation policies to innovative kernel structures) that can be used to increase the temporal predictability of a system, and might be useful in this situation. Most of those techniques have been applied to the experimental setup described in the previous sections, to test them and to evaluate their performance. However, in the following of this section only the solutions that resulted to be really effective are described.

### A. The Real-Time Kernel

First of all, a real-time version of the Linux kernel (called Preempt-RT [27]) has been tested.

The goal of a real-time kernel is to reduce the worst-case latencies normally experienced when a non-real-time kernels is used (thus making the system more predictable). In contrast to general-purpose kernels, real-time kernels do not care about average performance (or average throughput), but are designed to optimise the worst-case performance — in this case, to reduce the maximum kernel latencies. In general-purpose OSs, such kernel latencies are due to non-preemptable sections in the kernel, used to avoid race conditions between concurrent system calls, or between applications' code and interrupt handlers.

A real-time kernel developed from scratch is explicitly designed to avoid large non-preemptable sections, so that the kernel latencies can be kept under control. On the other hand, some real-time kernels have been developed by modifying existing non-real-time kernels such as Linux. In this case, the "traditional" general-purpose kernel structure (generally derived from the original "monolithic" UNIX structure) is somehow modified to reduce the size of non-preemptable sections without introducing the race conditions that such non-preemptable sections are designed to avoid. Some solutions, such are RTLinux [34], RTAI [22], or Xenomai [17] make a distinction between non-real-time applications (which can use the functionalities provided by the Linux kernel) and real-time applications, which can only use some reduced functionalities provided by a specialised kernel. Such a specialised real-time kernel has direct access to the hardware, and runs *below* the Linux kernel (this is why this kind of approach is generally described as "dual kernel"). The Linux kernel cannot directly access the hardware, but accesses it by passing through the real-time kernel. As a result, the critical sections of the Linux kernel only needs to be non-preemptable for the non-real-time applications, while can be preempted by the real-time applications.

The dual kernel approach achieves very small kernel latencies for real-time tasks, but does not allow such tasks to access the functionalities provided by the Linux kernel. Hence, to implement TDP using this approach a new driver for the network card has to be written, and the routing/forwarding algorithms provided by the Linux kernel have to be re-implemented. For this reason, a different approach to real-time kernels has been used in this paper. In particular, the worst-case end-to-end latencies have been reduced by using Preempt-RT, which modifies the standard Linux kernel to provide low latencies to *all* of the tasks running in the system.

Preempt-RT reduces the size of the kernel non-preemptable sections by transforming the ISRs and software interrupts into threads, named *hard IRQ threads* and *soft IRQ threads*. Once such interrupt handlers are transformed into threads, proper mutexes are used to handle many of the critical sections. In a non-real-time kernel, an ISR always interrupts the current execution, and cannot be interrupted by higher priority activities. In contrast, Preempt-RT uses dedicated threads for serving the interrupt requests: in this way, the priority of an IRQ thread can be lowered not to interfere with important real-time activities, or increased to reduce the interrupt latency. Moreover, IRQ threads are preemptable schedulable entities, like all the other threads. In addition, non-real-time kernels generally protect critical sections inside the kernel by making them non-preemptable. Preempt-RT, instead, uses mutexes to protect most of such critical sections. Finally, Preempt-RT introduces Priority Inheritance [29] on the mutexes to make them more predictable and reduce worst-case latencies. Thanks to these properties of Preempt-RT, no modifications to the interrupt handlers are necessary to reduce the end-to-end latencies, and a predictable software implementation of TDP can be realised by adjusting the priorities and CPU affinities of the various system threads (including IRQ threads). Finally, notice that since the ISR of the GPS IRQ runs in interrupt thread the packet transmission operations can be directly invoked by such an ISR (instead of triggering a software interrupt which will be in charge of transmitting the packets).

### B. APIC Interrupt Latency

Before starting to optimise the various components of the end-to-end latency, it is important to check if the used hardware has the potentiality to support a TDP router. To this purpose, the `cyclictest` program has been used to measure the interrupt latency when a hardware timer (specifically, the APIC timer) is programmed to generate an interrupt every $100\mu s$ (since $TF = 100\mu s$ is the target time-frame size).

Each experiment was $500s$ long, and some additional workload, referred to as *kernel stress*, has been introduced in background, to increase the probability to measure high latencies. Such a kernel stress is composed by the `hackbench`[5] and `cache calibrator`[6] programs, which are known to trigger high latencies[7].

All the experiments presented in this section have been performed on a GNU/Linux system based on the Ubuntu 8.04 (Hardy) distribution. The used kernels are a vanilla 2.6.29 (non real-time) and a Preempt-RT `2.6.29.5-rt22` (real-time) compiled from source. Since Preempt-RT is able to reduce the kernel latencies even in presence of a significant user-space workload, it is not necessary to boot the OS in single-user mode.

When using the vanilla 2.6.29 Linux kernel, the average interrupt latency resulted not to exceed $6\mu s$, but the worst-case interrupt latency measured in a $500s$ long experiment was $2434\mu s$.

According to these results, a non-real-time kernel is not suitable for efficiently implementing a TDP router with a small time-frame. The experiment has been repeated using the `2.6.29.5-rt22` real-time version of the Linux kernel: The maximum interrupt latency measured by `cyclictest` resulted to be $25\mu s$. This huge improvement came at a small cost: increasing the average interrupt latency to $7.05\mu s$.

As a conclusion, `cyclictest` results suggest that the used hardware can meet the real-time requirements for the TDP router (if a proper software stack is used).

### C. GPS Card Interrupt Latency

After verifying that the hardware can support a TDP router with small time-frames (if a proper real-time kernel is used), the interrupt latency experienced when a GPS card is used as an interrupt source has been measured and reduced. To this purpose, the same GPS card used for the experiments reported in Section III-D (a Symmerticom bc635PCI-U) has been connected to the test machine and adopted to generate the periodic interrupt. The card has been programmed to generate an interrupt every $100\,\mu$s (which, as said above, is considered the target time-frame size $TF$ in this work, and was non sustainable when using a non-real-time kernel - as shown in Figure 2). A custom driver has been written for the GPS card, and has been used to measure the GPS IRQ latency, as explained in Section III-D. Surprisingly enough, the worst-case GPS IRQ latency ($71\mu s$) resulted to be much larger than the worst-case interrupt latency measured in Section IV-B. This result was found to be caused by the fact that `cyclictest` uses the APIC timer (which is directly connected to the CPU) as an interrupt source, while the GPS card is connected to the CPU through the PCI bus.

By disabling one of the two CPUs (thus transforming the SMP machine in a uniprocessor), the worst-case GPS IRQ latency decreased to $41\mu s$. Since this fact seemed to suggest that part of the problems were due to the multiprocessor nature of the test machine, some techniques used in multiprocessor real-time systems were investigated. Among such techniques, the so called *CPU Shielding* [9] proved to be very effective.

CPU Shielding is designed to reduce the interference on real-time tasks in multiprocessor systems: one of the system CPUs is dedicated to real-time activities, which run on it and cannot migrate on other CPUs. Similarly, non-real-time applications cannot migrate on the dedicated CPU, and generic kernel activities (not needed by real-time applications) cannot be executed on it. In this way, real-time activities do not suffer from any interference from non-real-time activities, and the latencies experienced by real-time tasks can be reduced. While implementing CPU Shielding on a general-purpose kernel can be difficult, it can be easily implemented on Preempt-RT by binding the IRQ threads needed by real-time applications (the GPS IRQ thread, in this case) to the dedicated CPU and by preventing all the other threads and processes from executing on it. In the following tests, the GPS IRQ thread was bound to CPU 1, whereas all the other threads and processes were bound to CPU 0. Moreover, the IRQ affinity mask has been configured to deliver the GPS interrupt requests to CPU 1, and all the other interrupt requests to CPU 0. Note that dedicated threads for capturing network packets and thread/IRQ affinity settings have been used in other works [16] to improve the received throughput in packet capturing. The solution used in this work is slightly different, using one single interrupt thread - provided by Preempt-RT - to receive the packets, because the main goal is to decrease the worst-case latency and not to improve the throughput.

To further reduce the interrupt latency experienced by the GPS IRQ thread, its priority has also been increased to the maximum possible value. With such a tuned configuration, Preempt-RT was able to reduce the worst-case GPS IRQ latency to $19\mu s$ (even when the previously described kernel stress was used to trigger high latencies). The length of the experiments has been increased to 12 hours, but this did not result in any increase to the worst-case GPS IRQ latency. These results indicate that to optimise the GPS IRQ latency Preempt-RT has to be properly configured.

---

[5] http://git.kernel.org/?p=linux/kernel/git/clrkwllms/rt-tests.git

[6] http://homepages.cwi.nl/~manegold/Calibrator/

[7] https://rt.wiki.kernel.org/index.php/Worstcase_Latency_Test_Scenario

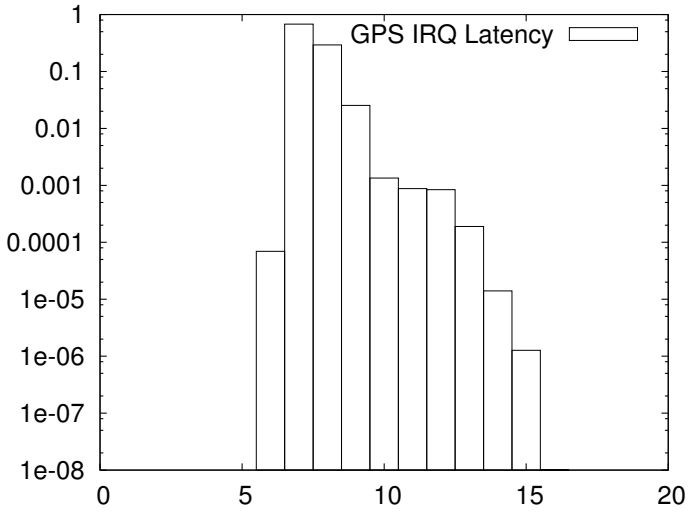


Fig. 3. Probability Mass Function of the interrupt latency $L^I$ for the GPS interrupt.

After evaluating the worst-case GPS IRQ latency without considering the packet transmission, an optical Ethernet card has been connected to the system, to measure the transmission latency. However, the simple fact of connecting a new card to the PCI bus resulted in a huge increase in the GPS IRQ latency (which should not have been affected by the new PCI card). This issue resulted to be related to a PCI bus arbitration problem, and has been solved by changing the interrupt lines used by the various cards. Surprisingly enough, this change also resulted in a small decrease of the worst-case interrupt latency, which was measured as $16\,\mu$s in another 12 hours long test. The PMF of the resulting GPS IRQ latency is displayed in Figure 3 (note that the probability to have $L^I = 16\mu s$ is $1.0046 \times 10^{-8}$, which is too small to be visible in the figure.

### *D. Full End-to-End Latency*

To measure the end-to-end latency, the configuration presented in Section III-B has been used.

Preliminary measurements of the end-to-end latency $L$ did not provide the expected, so that some additional investigation about the various latency component was needed. First of all, the inter-packet times have been measured to check if the experimental setup was working correctly. This experiment resulted in inter-packet times ranging from $50$ to $150\mu s$, instead of the expected constant value of $100\mu s$, and provided some hints about the bad end-to-end latencies observed. Figure 4 reports the inter-packet Times (in $\mu$s) for the first $30000$ received packets (3s), showing the problem.

Simple tests proved that the random latency was not on the sender side, but was part of the receiving latency $L^{Rx}$. Since the interrupt latency on the receiver was measured to be smaller than $16\mu s$ (like the interrupt latency on the sender), the additional receiving latency (more than $35\mu s$) was probably generated by the network card. After some investigation, it turned out that such an unpredictability in the receiving latency was due to the *interrupt mitigation* (or *interrupt coalescing*) feature of the network card. Under heavy network load, the card does not generate one interrupt request each time a packet is received, but tries to serve multiple packets with the same interrupt. In this way, the overhead caused by the interrupt service routines can be reduced by "clustering" some of the interrupts requests [12], [20] and receiving multiple packets with one single invocation of the interrupt service routine[8].

Fortunately, the network card allows disabling the interrupt mitigation mechanism, and after disabling this feature the Inter-Packet Times returned to be acceptable (see Figure 5). Longer experiments confirmed that after disabling interrupt mitigation in the network card, the problem highlighted above disappears. Of course, disabling interrupt mitigation has a cost: in particular, it increases the amount of CPU time consumed by the network card interrupt handler. More measurements revealed that in a `ping -f` test (i.e., a network load close to the worst-case situation) the percentage of CPU time consumed by the network driver increased from $8.48\%$ to $10.70\%$ when disabling interrupt mitigation. This is an acceptable cost for having a predictable receiving latency.

Note that some of the techniques used in this paper for reducing the variations in the Inter-Packet Times (namely, the usage of the Linux Preempt-RT kernel and the correct configuration of the IRQ thread priorities) have been used in previous works to reduce the jitter in an EtherCAT device [11]. The maximum difference between the measured Inter-Packet Times and the

[8]Note that in the cited papers interrupt mitigation is implemented in software, whereas in our test system it was implemented in hardware by the NICs.

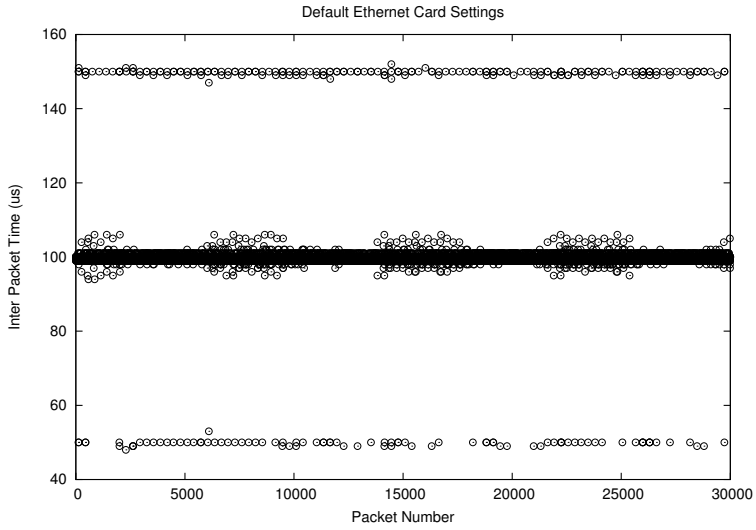


Fig. 4. Inter-Packet Times (in $\mu s$) measured in $3s$ with the original network card settings.

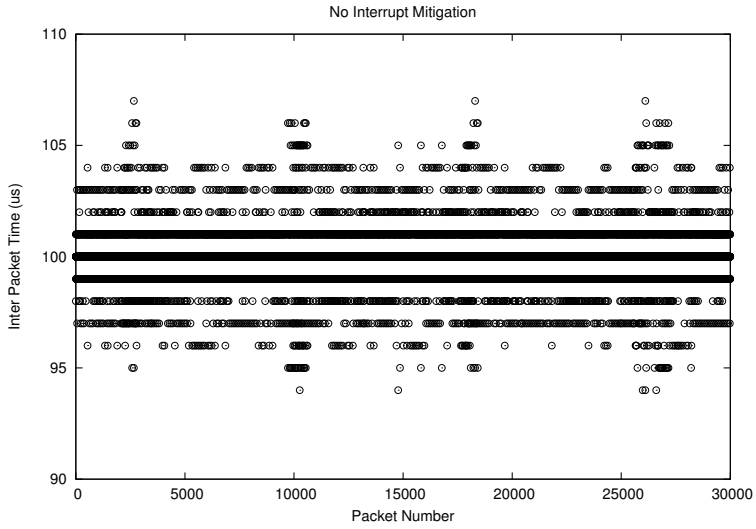


Fig. 5. Inter-Packet Times (in $\mu s$) measured in $3s$ with the original after disabling interrupt mitigation for the network card. Note that with the original settings the Inter-Packet Times arrive to more than $150\,\mu$s, whereas after disabling interrupt mitigation the receiving latency is under control.

expected value $TF = 100\mu s$ is less than $10\mu s$, which is comparable with the worst case jitter measured in [11]. However, in this paper the goal is not to reduce the jitter, i.e., the variation in packet inter-arrival time, but to minimize the worst case end-to-end latency. Furthermore, we deal with a synchronous technology and we have to care about its proper operation. This was not considered in [11] as EtherCAT operates asynchronously. Hence additional techniques (e.g., interrupt shielding, synchronisation with the GPS IRQ thread) are used to achieve this goal. The fact that our results are comparable with that presented in [11], regardless of scope and experimental setup, is an indirect validation of both works.

Once the receiver has been properly configured, it was possible to measure the sum of the transmission latency and the receiving latency, $L_b^{Tx} + L^{Rx}$. This measurement has been performed by modifying the transmitter to write the current GPS time $t^1$ in each packet, immediately before sending it. The receiver can then measure the GPS time $t^2$ when the packet is received, and compute $L_b^{Tx} + L^{Rx} + \delta = t^2 - t^1$. Note that the propagation and transmission delay $\delta$ is smaller than $1\mu s$ (much smaller than $L_b^{Tx}$ and $L^{Rx}$, hence it can be ignored in this case; as a result, it was considered $L_b^{Tx} + L^{Rx} = t^2 - t^1$. The PMF of such latencies in a $4$ hours long experiment is reported in Figure 6. As it is possible to see, the maximum value is $41\mu s$. Since the worst-case interrupt latency $L^I$ is $16\,\mu$s, the worst-case end-to-end latency $L$ can be expected to be $L^I + L_b^{Tx} + L^{Rx} = 16 + 41 = 57\mu s$.

Finally, we measured the end-to-end latency $L$ to compare it with the values computed above ($57\mu s$). The end-to-end latency

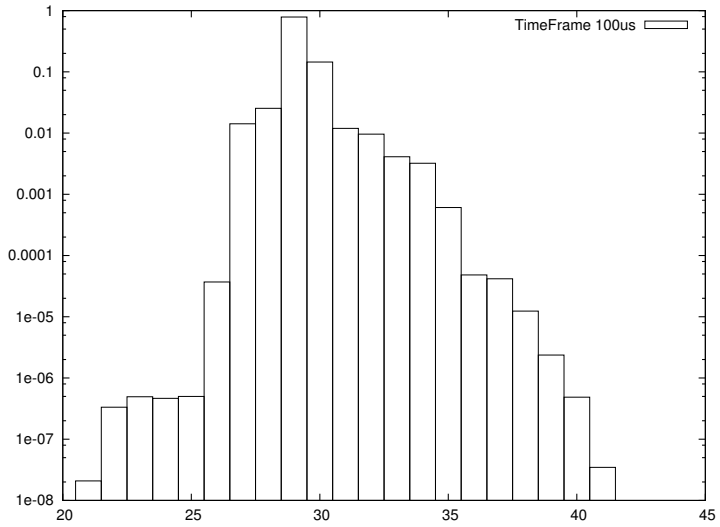


Fig. 6. Probability Mass Functions of the transmission and receiving latencies using the Linux kernel and Preempt-RT.

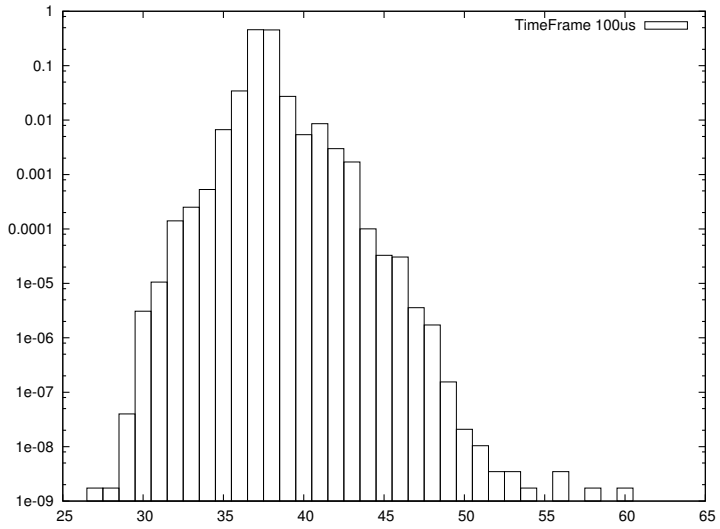


Fig. 7. Probability Mass Functions of the end-to-end latencies using the Linux kernel and Preempt-RT.

has been measured in the same way as it was measured for the experiments reported in Figure 2. Figure 7 plots the PMF obtained with the Preempt-RT version of the Linux kernel in a 16 hours long experiment. From the figure, it is possible to notice that the worst-case end-to-end latency is $60\mu s$, which is compatible with the expected value of $57\mu s$ (see above). The minimum end-to-end latency is $25\mu s$, which can be compensated generating the GPS IRQs in advance with respect to the beginning of the time-frame.

These results prove the high gain achievable by using real-time techniques. Summing up, by comparing Figure 7 and Figure 2 when $TF = 100\mu s$ it is immediately possible to appreciate the advantages of using a real-time kernel.

## V. Discussion and Conclusion

This paper analysed the different components that form the latency on both the transmitting and receiving side of standard Internet hosts (UNIX-based) and edge routers, which are also normally based on UNIX or UNIX-like operating systems with the goal of implementing a gateway between standard Internet systems and time synchronised systems. Measurements on an existing implementation showed that the behaviour of non-real-time OSs (which are optimised for throughput and performance rather than for meeting real-time targets) cannot meet the requirements, i.e., that jitter and worst case delays due to interrupt

handling both in the kernel, CPU and on the bus transferring packets to the NIC card are too large for synchronised operations, leading to large overheads.

The real-time Linux-based implementation tested in this work is instead able to reduce the worst case end-to-end (i.e., including transmission and reception) latency to a level that makes it feasible to use timeframes of $100\mu s$ to $200\mu s$ with an acceptable overhead. All of the software used in this paper is released under an open-source license (the GPL) and is publicly available.

A direct measure of both end-to-end and transmission latencies requires the development of specialised hardware, synchronised with the Common Time Reference (GPST), able to 'sniff' the packets on ingress or egress the optical transmission links or in other suitable points. The development of this hardware is difficult and expensive, thus we resorted to measurements based on the same machines that are involved in the transmission and reception of packets. This task is non-trivial because the latencies to be minimized and measured and are of the same order of magnitude of the accuracy of the timing functions provided by the OS kernel. This situation requires special attention to avoid mixing measures' noise with pseudo-random variation of the measured quantity, which are the object of minimization.

As a future work, the performance of a full TDP gateway (including packet classification and routing) will be evaluated. A simple synchronised router (able to receive traffic on a standard Ethernet interface and send it properly synchronised on an interface equal to the one analysed in this work) will be used for this purpose. Furthermore, the use of virtual interfaces will be explored as a means to separate different traffic types in different timeframes, or use different timeframes for routing purposes, i.e. transmitting traffic with different destinations in different timeframes, so that a TDS (Time Driven Switching) backbone can operate with all the benefits of pipelined forwarding. Finally, the possibility to take advantage of modern NIC designs, providing multiple packet queues [16] will be considered. This feature, combined with a multi-core CPUs and a more advanced configuration of the CPU shielding technique could allow to improve the router performance.

A different solution can be based on dedicated hardware, in practice integrating a GPS card with an Ethernet card, so that the interrupts from the GPS-driven clock can be used directly in hardware, thus eliminating some sources of latencies. Although this solution is appealing and appears elegant, it does have a number of drawbacks. First of all developing such a specialised card is not trivial and quite expensive. Second, and more important, this solution appears not too difficult to design if the only goal is sending the largest number of enqueued packets per timeframe. If instead some further functionality is required, like classification, tagging, routing, or any other operation that requires complex processing, then the 'help' of a general purpose machine is required in any case, which means that the hardware-synchronised card must work together with the hosting machine, and coordinate with the kernel via the PC bus, which in practice brings the situation back to the one we have analysed and optimised in this work.

## REFERENCES


[1] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the IEEE Real-Time Embedded Technology and Applications Symposium*, San Jose, California, September 2002.

[2] D. Agrawal, M. Baldi, M. Corra, G. Fontana, G. Marchetto, V.T. Nguyen, Y. Ofek, D. Severina, T.H. Truong, and O. Zadedyurina. A scalable approach for supporting streaming media: Design, implementation and experiments. In *12th IEEE Symposium Computers and Communications, 2007 – ISCC 2007*, pages 211 –217, July 2007.

[3] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Trans. Dependable Secur. Comput.*, 4:180–190, July 2007.

[4] M. Baldi and O. Ofek. Time for a "greener" internet. In *IEEE Int. Workshop on Green Communications (GreenComm) at ICC 2009*, June 2009.

[5] Mario Baldi and Riccardo Giacomelli. Time-driven access and forwarding in ieee 802.11 mesh networks. In *Proceedings of the 2009 Second International Conference on Advances in Mesh Networks*, MESH '09, pages 12–18, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Mario Baldi and Guido Marchetto. Pipeline forwarding of packets based on a low-accuracy network-distributed common time reference. *IEEE/ACM Trans. Netw.*, 17:1936–1949, December 2009.

[7] Mario Baldi, Guido Marchetto, Fulvio Risso, Giulio Galante, Riccardo Scopigno, and Federico Stirano. Time driven priority router implementation and first experiments. In *Proceedings of the IEEE International Conference on Communications (ICC 2006)*, Istanbul, Turkey, June 2006.

[8] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle. Comparing and improving current packet capturing solutions based on commodity hardware. In *Proceedings of the 10th annual conference on Internet measurement*, pages 206–217. ACM, 2010.

[9] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard linux. In *Proceedings of Fourth Real-Time Linux Workshop*, Boston, MA, December 2002.

[10] E. Carlson, C. Prehofer, C. Bettstetter, H. Karl, and A. Wolisz. A distributed end-to-end reservation protocol for ieee 802.11-based wireless mesh networks. *Selected Areas in Communications, IEEE Journal on*, 24(11):2018 –2027, nov. 2006.

[11] M. Cereia, I.C. Bertolotti, and S. Scanzio. Performance evaluation of an ethercat master using linux and the rt patch. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, pages 1748–1753. IEEE, 2010.

[12] Xiaolin Chang, J.K. Muppala, Zhen Han, and Jiqiang Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *Proceedings of the IEEE International Conference on Communications (ICC'08)*, pages 1835–1839. IEEE, 2008.

[13] Luca Chiaraviglio, Marco Mellia, and Fabio Neri. Reducing power consumption in backbone networks. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC'09, pages 2298–2303, Piscataway, NJ, USA, 2009. IEEE Press.

[14] Ken Christensen, Bruce Nordman, and Rich Brown. Power management in networked devices. *Computer*, 37:91–93, August 2004.

[15] Ed. E. Mannie. Generalized multi-protocol label switching (gmpls) architecture, October 2004.

[16] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of IMC 2010*, 2010.

[17] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on gnu/linux, 2004.

[18] M. Guenach, C. Nuzman, J. Maes, and M. Peeters. On power optimization in dsl systems. In *IEEE Int. Workshop on Green Communications (GreenComm) at ICC 2009*, June 2009.

[19] Maruti Gupta and Suresh Singh. Greening of the internet. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 19–26, New York, NY, USA, 2003. ACM.

[20] Ilhwan Kim, Jungwhan Moon, and Heon Y. Yeom. Timer-based interrupt mitigation for high performance packet processing. In *Proceedings of the 5th International Conference on High Performance Computing in the Asia-Pacific Region, Gold*, 2001.

[21] Chung-Sheng Li, Y. Ofek, and Moti Yung. "Time-Driven Priority" flow control for real-time heterogeneous internetworking. In *Proceedings of the 15th Annual Joint Conference of the IEEE Computer Societies (INFOCOM '96)*, volume 1, pages 189 –197 vol.1, March 1996.

[22] P. Mantegazza, EL Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, 2000(72es):10–es, 2000.

[23] Viet-Thang Nguyen, Renato Lo Cigno, and Ofek Yoram. Tunable laser-based design and analysis for fractional lambda switches. *Communications, IEEE Transactions on*, 56(6):957–967, June 2008.

[24] Jeyashanker Ramamirtham and J. Turner. Time sliced optical burst switching. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 2030–2038, March 2003.

[25] P. Regnier, G. Lima, and L. Barreto. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Operating Systems Review*, 42(6):52–63, 2008.

[26] K. Ross, N. Bambos, K. Kumaran, I. Saniee, and I. Widjaja. Scheduling bursts in time-domain wavelength interleaved networks. *Selected Areas in Communications, IEEE Journal on*, 21(9):1441–1451, November 2003.

[27] Steven Rostedt. Internals of the rt patch. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2007.

[28] F. Schneider and J. Wallerich. Performance evaluation of packet capturing systems for high-speed networks. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 284–285. ACM, 2005.

[29] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[30] Thu-Huong Truong, M. Baldi, and Y. Ofek. Efficient scheduling for heterogeneous fractional lambda switching (fls) networks. In *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 2331 –2336, nov. 2007.

[31] K. Ueda and T. Yakoh. Development of time division switching hub for synchronous tdma. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 45 – 50, sep. 2004.

[32] J.Y. Wei and Jr. McFarland, R.I. Just-in-time signaling for wdm optical burst switching networks. *Lightwave Technology, Journal of*, 18(12):2019 –2037, December 2000.

[33] T Yakoh. Synchronous time division internet for time-critical communication services. In *EUROMICRO International Workshop on Real-Time LANs in the Internet Age*, pages 111–114, June 2002.

[34] V. Yodaiken and M. Barabanov. A real-time linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. Citeseer, 1997.

[35] O. Yu and Ming Liao. Synchronous stream optical burst switching. In *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, pages 1447 –1452 Vol. 2, October 2005.