Chasing Linux Jitter Sources for Uncompressed Video

Arthur Toussaint*†, Mohammed Hawari*†, Thomas Clausen*
*Ecole Polytechnique, France, †Cisco Systems, France
{arthur.toussaint, mohammed.hawari, thomas.clausen}@polytechnique.edu

***Abstract*—Beyond the transport of uncompressed video over IP networks, defined in standards such as ST2022-6, the ability to build software-based Video Processing Functions (VPF) on commodity hardware and using general purpose Operating Systems is the next logical step in the evolution of the media industry towards an "all-IP" world. In that context, understanding the jitter induced on an ST2022-6 stream by a commodity platform is essential. This paper describes a general methodology to enumerate jitter sources on commodity platforms and to quantify their relative contribution to the overall system jitter. The methodology is applied to the Linux kernel, producing a classification of the different sources of jitter, and a quantification of their impact.**

I. Introduction

Live video processing in the broadcasting industry has historically been done using dedicated hardware appliances, interconnected by Serial Digital Interface (SDI) family interfaces *e.g.,* ST292M [1] for SDI over coaxial cable for HD video streams. Development of protocols such as ST2022-6 and ST2110-20 [2], [3], specifying encapsulation of SDI data over Ethernet/IP networks, has since 2012 indicated a trend towards replacing dedicated appliances with commodity hardware and networks, to gain operational flexibility and to reduce cost. These protocols — used in the video *production* world — differ from those used by end-users on the Internet — the video *distribution* world — since the former encapsulate raw video data whereas the latter rely on compression: the nominal rate of an ST2022-6 stream is never lower than $1.5\,\mathrm{Gbit/s}$, compared to the tens of $\mathrm{Mbit/s}$ for a compressed distribution HD stream. These standards for video production streams require the sender to emit packets at a Constant Bitrate (CBR). Also, while distribution streams can tolerate delays in the order of seconds, to allow for the interactivity required for broadcasting operations, *e.g.,* video stream mixing, production streams bound the delay at a few milliseconds.

This informs packet buffer dimensioning on video production appliances: more buffering at the ingress of a device, yields a greater delay. Thus, ST2022-6 receivers have small buffers – a side-effect of which is, that production streams cannot accommodate for much, if any, jitter. Thus, software-based Video Processing Functions (VPF) executed on Commodity Off-The-Shelf (COTS) servers need to minimize the jitter they introduce as much as possible. As any VPF needs to receive packets first, it is necessary to understand what minimal jitter the Operating System (OS) and its network stack introduce.

A. Related Work

Understanding jitter on general-purpose OS'es has, especially, been studied for real-time or High-Performance Computing (HPC) applications. OS jitter quantifies how unpredictable the performance of a running application will be. An experimental analysis of the effects hereof on CPU-bound tasks in a distributed HPC environment is given in [4] – which shows that jitter affects the overall performance of multi-stage workloads, where each stage is running on parallel nodes. Specifically, jitter significantly impacts the synchronisation steps between each stage, incurring a significant waste of computing capacity. In-kernel methods to quantify accurately the contribution of each jitter source to the overall system jitter are developed and evaluated in [5], [6].

For *hard real-time applications*, a deterministic lower bound on the performance is required. A recurring problem is determining the variability of *the response time i.e.*, the total elapsed time from when an interrupt request is raised, and until the corresponding application-level thread is scheduled. From this perspective, [7] compares Real-Time Operating Systems (RTOS) and general purpose OS'es, in the context of embedded systems used in experimental nuclear physics.

Aside from the analysis in [8] of periodic networked systems with events in the order of $100\,\mu\mathrm{s}$ on a FreeBSD-based COTS server, little attention has been given to characterising jitter on periodic events.

Yet, with ST2022-6 prescribing a CBR stream giving rise to a packet arrival time with a periodicity in the order of $7.41\,\mu\mathrm{s}$, if a VPF is to be successfully executed on a COTS server, a granular understanding of its jitter properties is required.

B. Statement of Purpose

This paper characterises the jitter, introduced by a COTS x86 server running a Linux-based operating system, upon reception of network packets corresponding to an ST2022-6 video stream. This includes an analysis of the packet reception path in the Linux kernel, an enumeration of identified jitter sources, and an experimental quantification of the relative contribution of each of these.

C. Paper Outline

The remainder of this paper is organised as follows: section II describes the data-path taken by a packet, *from wire to application*, enumerating the potential sources of jitter that can be encountered. Section III motivates and introduces the

978-3-903176-14-0

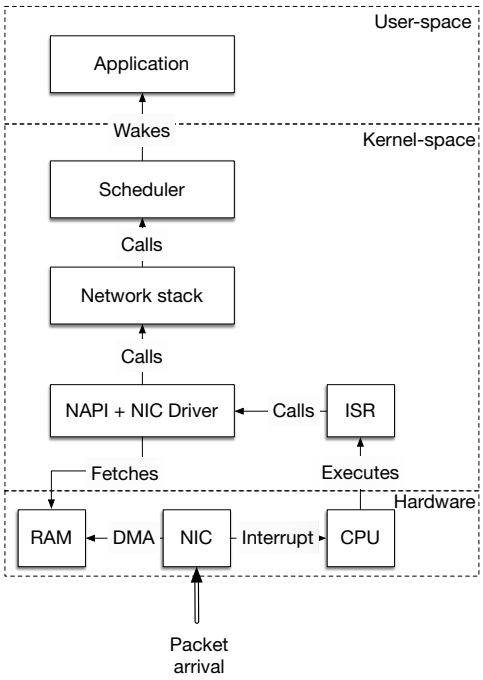


Figure 1. Schematic view of the path taken by a data packet – from Network Interface Card (NIC) to Application.

experimental setup used to quantify these sources of jitter, which is then used for producing the results presented in section IV. This paper is concluded in section V.

II. From wire to application

The jitter sources along the path of a packet through a COTS server, depicted in figure 1, from its arrival at the Network Interface Card (NIC) until it is delivered to an application, are analysed in this section.

A. From Wire to Interrupt

When a packet arrives at the NIC, it is decoded and copied into RAM using *Direct Memory Access* (DMA). DMA allows external devices, such as NICs, controlled access to a portion of the CPU's RAM.

DMA uses the system PCI bus, which is a shared resource with potential contention for access – and hence, is a potential source of jitter. Another source of jitter is access to the RAM itself, since the NIC (hardware), and the NIC driver (part of the operating system) will be competing for access hereto. Finally, multiple layers of cache, which are shared with all the processes of a CPU, can introduce further jitter during the phases of copying data to and from RAM.

Once a packet has been copied into RAM, the NIC raises an interrupt to signal to the CPU that a new packet is available. Interrupts are also raised through the system PCI bus, where contention may again introduce jitter. However, some NICs also implement *Interrupt Rate Throttling* (ITR), which delays or suppress some interrupts from being raised, so as to avoid interrupt overload at high data rates. While this feature does reduce the OS per-packet processing cost, it does constitute an additional source of jitter, especially among packets received in a periodic stream. Illustrating this by a simple example, if ITR suppresses 9 out of 10 interrupts, then packet number 1 in a stream will incur a further delay of receipt of another 9 packets before an interrupt is raised, and it can be processed, whereas receipt of packet number 10 will cause an interrupt to be raised immediately.

B. From Interrupt to Application

A raised interrupt triggers a call to the kernel *Interrupt Service Routine* (ISR). The time from an interrupt is raised, and until the beginning of the execution of the ISR can vary, *e.g.,* due to other higher-priority or non-masked interrupts, or the need to awaken the core executing the ISR from suspension. Thus, this constitutes a potential jitter source.

Execution of the ISR is the first event, which can be timestamped in software by the operating system. In Linux, specifically, this is the `irq_entry` event. Then the ISR calls the *New API* (NAPI) component, which attempts to reduce the load induced by network activity on the CPU during high load scenarios, by processing packets in bursts. Thus, this also constitutes a potential jitter source.

NAPI calls the NIC driver, which fetches the packets from RAM (where they had been placed using DMA by the NIC) – and hands these off to the set of kernel components constituting the network stack, see figure 1, for further processing (decoding received packet headers, extracting metadata corresponding to the different network layers, etc). This processing is subject to optimisations such as memory prefetching, cache hits, etc., and therefore also constitutes a potential jitter source.

The processing step in the networking stack is to identify if a given packet matches an open socket – *i.e.,* if there's an application able to receive the payload of the packet. If there is, and if the application process is sleeping, or is waiting for data from this socket, it is awoken by the kernel – which requires (i) a call to the scheduler and (ii) a context switch. These two operations also constitute a potential jitter source.

C. Network-Independent Jitter

In addition to the jitter sources within the data-path itself other sources of jitter — henceforth *network-independent* jitter — exist. Essentially, those consists of events that temporarily interrupt packet processing anywhere on the path discussed in section II-B and illustrated in figure 1.

First, the Linux kernel's scheduler can preempt running processes. Suspending a running process from execution will cause jitter, as the process will not be able to perform any action during the time it is not scheduled.

Second, hardware interrupts take precedence over any other kernel-space or userspace task. Thus, a non-masked interrupt being raised will trigger the kernel ISR, interrupting any other execution on the CPU core charged with handling that interrupt. This can introduce jitter in any part of the stack – noting that a high-priority interrupt being raised can delay the execution of the ISR corresponding to a packet arrival.

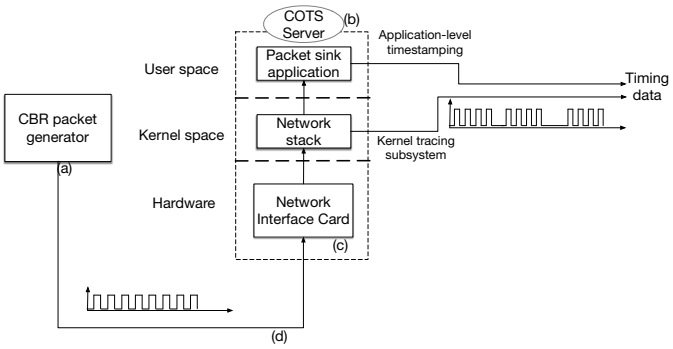


Figure 2. Abstract view of the analysed system

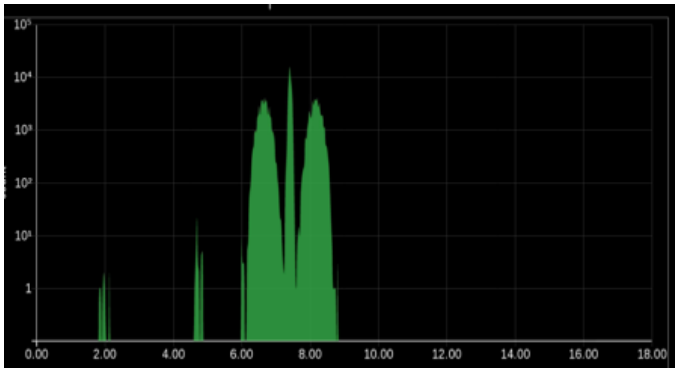
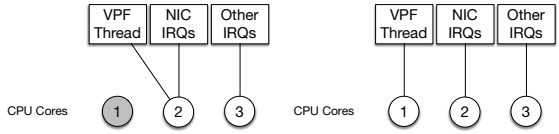


Figure 3. Histogram of the packet inter-arrival times at the ingress of the server



(a) Affinity in the baseline system

(b) Affinity in the system with re-routed IRQs

Figure 4. Interrupt and scheduling affinity. Involved cores are identified as core 1, 2 and 3.

Completely transparent to the operating system, System Management Interrupts (SMI) literally steal control of a CPU from the OS, for doing low-level house-keeping tasks. With no direct proof of their execution provided to the operating system kernel, SMIs are both a potential jitter source and are very hard to detect. One possible way to detect SMIs is to run an infinite loop polling the current time and to detect gaps in those measurements.

III. Experimental Setup

To quantify the contribution of the potential jitter sources, identified in section II, to the overall jitter of a VPF receiving an ST2022-6 stream, each is studied in an isolated environment.

A. A Packet Sink VPF

In order to eliminate any application impact (such as memory bandwidth consumption, CPU cache pollution, etc), a "packet sink VPF" with minimal application behaviour is used: on receipt of a packet, the application generates a timestamp, drops the packet without inspecting the payload, and computes the sequence of packet inter-arrival times ΔT. The resulting time series can then be analysed to quantify the jitter introduced by the server.

To differentiate between hardware-level and kernel-level jitter, another source of timestamps is needed – at the ingress of the kernel. For this purpose, the Linux kernel event tracing subsystem[1] is used, to record events at key steps of the packet data path. In particular, this allows recording timestamps for ISRs triggered by interrupts raised from the NIC, thus providing a second time series related to packet arrivals.

B. Quantitative Scope

The test setup is illustrated in figure 2, where the stream at the ingress of the server is **assumed** to be CBR. Understanding to what extent this assumption is true is **necessary**, to be able to interpret the recorded time-series **meaningfully**.

Thus the COTS server in figure 2 was substituted by an ST2022-6 hardware network analyser[2]. The measurement results are depicted in figure 3 and show a standard deviation $\sigma = 0.6\mu s$ in the distribution of the packet inter-arrival times. The stream received in the experimental setup is, therefore, CBR with a precision of $1\,\mu s$, and any sub-microsecond packet delay variation observed, therefore, cannot be attributed to the server hardware or software in this setup.

C. Hardware setup

The hardware setup is as follows, with reference to figure 2:

(a) A commercial SDI to ST2022 converter configured to output an ST2022 stream encapsulating a 1080i 29.97 frames per second video test pattern is used as **CBR Generator**.

(b) A server with two Intel(R) Xeon(R) CPU E5-2690 v4 is used as the **COTS server**.

(c) An Intel(R) XL710 with a $40\,\mathrm{Gbit/s}$ optical interface is used as the **Ingress NIC**.

(d) An interconnection between the packet generator and the COTS server through a Cisco Nexus 9000 fully non-blocking switch.

IV. Experiments and Results

This section experimentally quantifies the contribution of each source of jitter, as enumerated in section II. In the setup of section III, all known sources of jitter eliminated, a baseline is established. These sources are then restored one by one and their impact is measured.

A. Baseline: Minimal Jitter

To eliminate external jitter sources, some of the available CPU cores are isolated from the scheduler, and assigned statically and exclusively to executing the (user-space) VPF,

[1] https://www.kernel.org/doc/html/v4.17/trace/index.html

[2] Tektronix PRISM.

Table I
AVAILABLE KERNEL OPTIONS TO REDUCE NETWORK-INDEPENDENT JITTER

Kernel Option	Description
`nohz_full` = <CPU_LIST>	For each CPU core in <CPU_LIST>, disables the scheduler periodic tick when at most one thread is runnable on it.
`isolcpus` = <CPU_LIST>	Prevents the CPU cores in <CPU_LIST> from running any threads that were not explicitly assigned to a core in the list.
`rcu_nocbs` = <CPU_LIST>	Offloads Read Copy Update (RCU) callbacks from the CPU cores in <CPU_LIST> to a kernel thread scheduled elsewhere.
`rcu_nocb_poll`	Put the aforementioned kernel thread in polling mode so as to prevent RCU-offloaded CPU cores from having to notify the CPU core running that kernel thread.
`idle=poll`	Forces a polling idle loop, which reduces the time taken to wake up an idle CPU core by making it constantly busy.
`processor.max_cstate=1` `intel_idle.max_cstate=0`	Disables all energy-saving modes of the CPUs, further reducing the wake-up penalty.

handling NIC interrupts, and handling other (non-NIC) interrupts. This is done by way of using the Linux kernel options, indicated in table I, as follows:

```
nohz_full=<CPU_LIST> isolcpus=<CPU_LIST>
    ↪ rcu_nocbs=<CPU_LIST> rcu_nocb_poll
```

Moreover, as illustrated in figure 4a, the VPF is shielded from all interrupts other than those raised by the receiving NIC, as they are routed to a core different from the one running the VPF.

To eliminate jitter at the hardware level *i.e.,* between the arrival of the packet and the execution of the interrupt handler, the following is implemented. First, interrupt throttling is disabled in the `i40e` kernel module — the NIC driver used by the Intel XL710 — as follows:

```
ethtool -i <interface> -C adaptative-rx off
ethtool -i <interface> -C rx-usecs 0
```

Then, as illustrated in figure4a, the ISRs corresponding to the NIC are shielded from the rest of the interrupts of the system as these are routed to a CPU core different from the one executing the NICs ISRs. For each interrupt number `<INT>` and target CPU core `<CPU>`, rerouting interrupts is achieved as follows:

```
echo <CPU> > /proc/irq/<INT>/
    ↪ smp_affinity_list
```

All energy saving options are disabled by adding the following kernel options (described in table I):

```
processor.max_cstate=1 intel_idle.max_cstate
    ↪ =0
```

With the same purpose, all CPU cores are configured to run at their maximum frequency:

```
echo performance | sudo tee /sys/devices/
    ↪ system/cpu/cpu*/cpufreq/
    ↪ scaling_governor
```

In order to further reduce the CPU core wake-up penalty and as illustrated by figure 4a, the NICs interrupts are routed to the same CPU core as the one which the VPF is scheduled, which spares one CPU core wake-up and one Inter-Processor Interrupt (IPI), as the network stack (running on the same CPU core as the ISR) will not need to communicate with another CPU core when notifying the VPF.

Finally, the packet sink VPF used for the experiments and described in section III is implemented in polling mode; by calling `recvfrom` in a tight loop with the `MSG_DONTWAIT` flag, the VPF never blocks which eliminate the call to the scheduler evoked in section II.

B. Baseline: Experiments and Results

To differentiate the network-independent jitter defined in section II-C from the jitter introduced by the processing of incoming packets, a *dummy program* is implemented; it consists of a loop, busy waiting for $7.41\,\mu s$ and generating a timestamp at each iteration. Therefore, this dummy program has no interaction with the network stack and is able to provide measurements of the network-independent jitter of the system. In that setup, the sequence of timestamp should increase by $7.41\,\mu s$ at every iteration, unless the dummy program is somehow interrupted. In that case, the time series ΔT corresponding to the difference between a sampled timestamp and the next one in the loop would show some spikes in the same order of magnitude of the network-independent jitter.

Figure 5 shows the results obtained with the dummy program running for one million iterations — corresponding to $7.41\,s$ in the baseline configuration. Four spikes in the order of $15\,\mu s$ can be observed, which gives an idea about the minimum jitter that can be observed on such a system, independently from the network stack.

In that same setup, figure 6b depicts the time series of packet inter-arrival times as measured by the VPF, while figure 6a shows the time series of the duration between two consecutive ISR, this data being obtained with the kernel tracing subsystem. Confronting both figures as well as figure 5 suggests that the jitter seen by the VPF is a mixture of (i) network-independent jitter as shown by the similarity of the spikes in figure 5 and figure 6b, and (ii) network jitter as shown by the similarity of the $1\,\mu s$-wide noise around the $7.41\,\mu s$ average in figure 6a and 6b.

Figure 7a and figure 7b give finer-grained information about the jitter introduced by the network stack itself *i.e.,* from ISR to the VPF. For example, there is a small but noticeable amplification in the standard deviation between the distribution

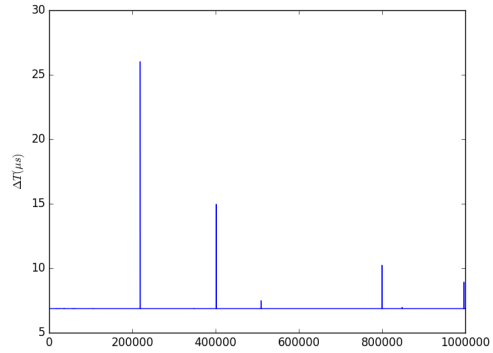
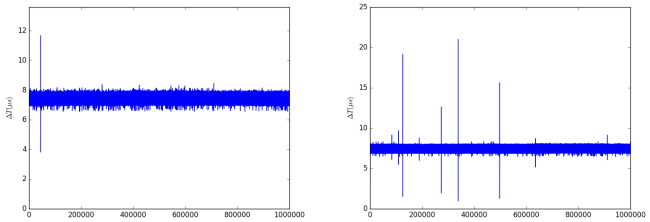
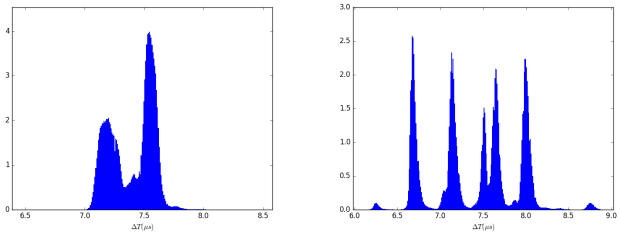


Figure 5. Dummy program : 1M acquisitions



(a) Tracing `irq_entry` (b) Timestamps in the VPF

Figure 6. Baseline system: time series



(a) Tracing `irq_entry` (b) Timestamps in the VPF

Figure 7. Baseline system: Histogram of ΔT

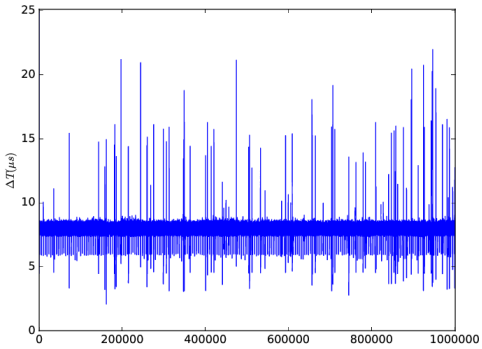


Figure 8. System without `idle=poll`: tracing `irq_entry`

of ΔT at the ISR level and the distribution at the VPF-level which corresponds to the jitter introduced by the network stack. Moreover, the clustering and discrete patterns observed in figure 7b can be plausibly explained by associating each cluster to a succession of events that happened during the packet processing. In other words, each cluster could correspond to a possible code path.

Given the previous analysis of the baseline system, the sources of jitter enumerated in section II are restored independently so as to study their relative contribution.

C. ISR Start Of Execution

For multiple reasons, the elapsed time between interrupt and ISR execution can vary, hence jitter. For example, the CPU core handling the interrupt can be idle when the interrupt is raised. This is evaluated by removing the `idle=poll` kernel parameter. In this situation, figure 8 shows the apparition of many spikes in the order of 15 µs when compared to figure 6a.

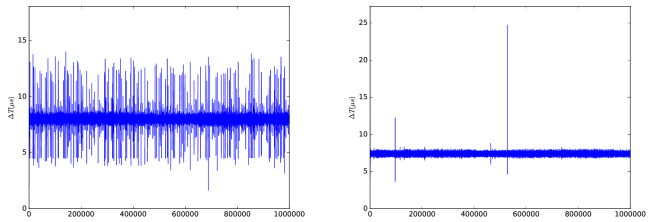
When interrupts are routed to a different CPU core than the one on which the VPF is scheduled, it is plausible to assume jitter reduction as ISRs are granted a dedicated core. Jitter increase is also plausible because of the requirement for inter-core synchronisation, which is a source of jitter *i.e.,* because of cache synchronisation or IPIs. Jitter increase is also possible because the newly dedicated core is not doing anything else, which means it is likely to be asleep, hence a wake-up-induced jitter at ISR execution.

To discriminate between those hypotheses, three experiments have been designed as follows: In the first experiment (figure 6a) NIC interrupts are routed to the same CPU core as the one on which the VPF is scheduled, in the second (figure 9a), NIC interrupts are *re-routed* to a different CPU core, and on the third (figure 9b), NIC interrupts are re-routed to a different CPU core, but the latter is kept busy by an infinite loop. The CPU core configuration in those two last experiments is illustrated in figure 4b. According to these figures, the most likely hypothesis is that routing the interrupts to a dedicated core slightly increases jitter in the absence of another program on the same core, and significantly increases it in the presence of an always runnable thread on that CPU core (with frequent spikes in the order of 15 µs).

D. Linux Scheduler induced jitter

The straightforward approach to packet reception is to use the `recvfrom` function exposed by the kernel, which in its default behaviour, and if no packet is available in the socket queue at the time of the call, blocks and triggers a context switch and a call to the Linux scheduler. It will, therefore, need to be rescheduled as soon as the network stack makes a packet available to the socket, hence introducing additional jitter. Even with the `idle=poll` kernel option, such an approach shows spikes of up to 16μs (figure 10. Without the latter kernel option, this blocking leads to even more jitter, showing spikes of up to 60μs

As explained in section IV-A, that jitter can be eliminated by receiving packets in polling mode.



(a) With infinite loop (b) Without infinite loop

Figure 9. System with re-routed interrupts: `irq_entry`

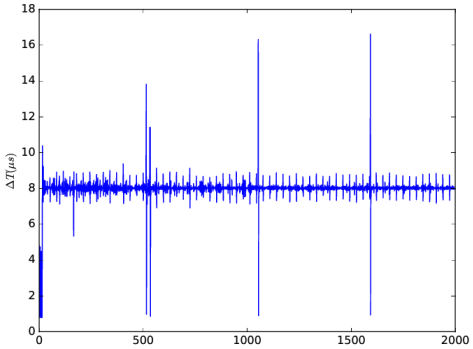


Figure 10. Measurements in blocking mode: VPF

E. Interrupt Throttling jitter

The major source of jitter studied in this paper originates in the NIC's interrupt moderation capacities. Those features prevent the NIC from flooding a CPU core with interrupts at high data rates by limiting the rate at which it triggers an interrupt. But at low data rates, those features introduce unnecessary and variable latency, as some interrupts are delayed, preventing the CPU from processing the incoming packet unless the throttling period has elapsed.

As described in [9], the Intel XL710 NIC, supports two interrupt moderation features: Interrupt Throttling (ITR) and Interrupt Rate Limiting (INTRL). ITR limits the instantaneous interrupt rate, and guarantees a minimum gap between two consecutive IRQs, whereas INTRL limits the average number of interrupts per second on a given period.

On the vanilla 4.13 Linux kernel, those options were not configurable at runtime and needed a kernel compilation in order to be changed. The default behaviour is to use an adaptive algorithm to change the ITR period depending on the current input bandwidth. On the 4.17 kernel, ITR is configurable using `ethtool` but INTRL still remains always-on. The results described hereafter were obtained with a custom kernel, specifically tweaked so as to disable INTRL.

The effects of ITR is quantified in this section by enabling it on the 4.17 kernel and choosing an ITR period equal to 100 µs. This value is chosen because it is the same as the default value on Linux 4.13. Results of this experiment are depicted in figure 11.

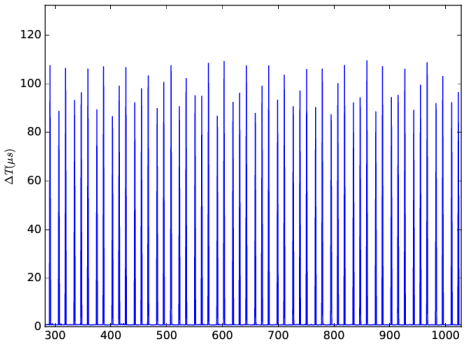


Figure 11. Measurements with ITR activated and setting of $T_{ITR} = 100\mu s$

The 100 µs spikes caused by ITR can easily be observed in this figure. These processing phases show nearly no jitter and do not show any pattern change if a blocking socket is used instead of a nonblocking polling loop. This last behaviour can be explained, as even a `recvfrom` call in a blocking setup will block once in every 100 µs as shown in figure 11. Therefore, ITR hides all the other sources of latency studied here.

V. Conclusion

As video production streams have a low tolerance for jitter, characterising it on a COTS server running a general-purpose OS is a crucial step, on the path towards building VPFs for a software-based, all-IP, video production setup. A detailed analysis of the packet reception path showed that sources of jitter can be classified as either (i) hardware-related such as ITR or the variable delay between the ISR and the interrupt, (ii) network-stack related such as the impact of the Linux scheduler, or (iii) network-independent such as SMIs, unrelated interrupts, or any higher priority code stealing cycles from the CPU. A quantitative study assessed that, in the context of ST2022-6 reception, hardware-related effects and especially ITR have the stronger impact, as they easily hide the impact of the Linux scheduler. The experimental methodology exposed in this paper consists of building a baseline system with a minimal jitter (through system analysis and experimental iterations) and reinstating each potential source of jitter, to study its impact.

The jitter characterisation for ST2022-6 achieved here gives, at the same time, results about the particular hardware setup used in the performed experiments — allowing to better understand the constraints a VPF needs to satisfy and the performance it can get from the OS — and a generic methodology to reproduce the study on any commodity platform, potentially enabling the implementation of software-based VPFs in a variety of environments.

References

[1] "ST 292-1:2018 - SMPTE standard - 1.5 gb/s signal/data serial interface," *ST 292-1:2018*, pp. 1–20, April 2018.

[2] "ST 2022-6:2012 - SMPTE standard - transport of high bit rate media signals over ip networks (HBRMT)," *SMPTE ST 2022-6:2012*, pp. 1–16, Oct 2012.

[3] "ST 2110-20:2017 - SMPTE standard - professional media over managed ip networks: Uncompressed active video," *ST 2110-20:2017*, pp. 1–22, Nov 2017.

[4] E. Vicente and R. M. Jr., "Exploratory study on the linux os jitter," in *2012 Brazilian Symposium on Computing System Engineering*, Nov 2012, pp. 19–24.

[5] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 852–863.

[6] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2007, pp. 331–340.

[7] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 435–439, 2008.

[8] M. Welponer, L. Abeni, G. Marchetto, and R. L. Cigno, "Measuring and reducing the impact of the operating system kernel on end-to-end latencies in synchronous packet switched networks," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1315–1330, 2012.

[9] *Ethernet Controller X710/ XXV710/XL710 Datasheet*, Intel Ethernet Networking Division, 2 2018, rev. 3.5.