System and Networking Aspects of the Transition of High-Performance Applications from Dedicated to Commodity Hardware : the Example of Media Production for Professional Broadcast

Mohammed Hawari

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAX053

Thèse de doctorat

System and Networking Aspects of the Transition of High-Performance Applications from Dedicated to Commodity Hardware: the Example of Media Production for Professional Broadcast

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Réseaux, Informations, et Communications

Thèse présentée et soutenue à Palaiseau, le 6 juillet 2021, par

M. MOHAMMED HAWARI

Composition du Jury :

Nadia Boukhatem Professeur, Telecom ParisTech (LTCI)	Président
Laurent Toutain Professeur, IMT Atlantique (SRCD)	Rapporteur
Lars Eggert Directeur Technique, NetApp	Rapporteur
Colin Perkins Associate Professor, University of Glasgow	Examinateur
Thomas Clausen Professeur, l'École polytechnique	Directeur de thèse

System and Networking Aspects of the Transition of High-Performance Applications from Dedicated to Commodity Hardware: the Example of Media Production for Professional Broadcast

Mohammed Hawari

2020

Abstract

Due to the increasing performance offered by commodity servers and to the general availability of multi-gigabit Ethernet-based networking hardware, a growing number of performance-intensive and network-intensive applications are being migrated from dedicated to commodity hardware. Examples thereof include scientific computing and network-packet processing, historically implemented by dedicated super-computing clusters and by dedicated packet-processing hardware, respectively. However, media production for professional broadcast (*i.e.,* the process by which multiple audiovisual sources are mixed and processed, in real-time, to elaborate the audiovisual stream as it will be consumed by the final viewer) is still being implemented with dedicated hardware equipment, based on the Serial Digital Interface (SDI), an interconnection technology carrying the legacy of analog video. Despite an ongoing industrial effort to replace SDI with IP-based interconnection — as specified by the SMPTE 2022-6 and 2110 standards — the delay-sensitive nature of media production still challenges its total transition to software running on commodity servers. This thesis solves different aspects of that problem.

First, the high rates and low jitter-tolerance of media production packet streams have motivated a quantitative and qualitative study of the sources of jitter undergone by those streams when they are processed by commodity servers. In addition to results specific to Linux x86_64 servers, that work has yielded a general jitter exploration methodology, applicable to any operating system and hardware commodity servers. Second, a generic platform enabling the implementation of custom high-accuracy instrumentation for hardware-based packet timestamping has been developed. By exposing a high-level programming interface — relying on the P4 language — that platform, despite being FPGA-based, allow network and broadcast operators with little hardware design skills to specify custom logic for line-rate packet processing and timestamping. In particular, such instrumentation can be used to qualify the jitter properties of media production streams. Third, a system to perform packet-pacing — *i.e.,* the transmission of a constant-rate packet stream with negligible jitter — has been proposed. By exclusively but cleverly relying on commodity hardware, that work invalidates the common belief according to which software-based media-production is impossible on commodity servers (due to the jitter they introduce). The proposed system has been formally and experimentally proven to yield a jitter, conforming to the requirements of media production streams. Finally, a software framework easing the implementation of media-production applications has been developed. That framework relies on a separation between media processing

and media transport: the media processing logic receives and transmits full media frames (*e.g.,* video frames) from the media transport logic, which handles high-performance packet processing with techniques such as zero-copy and kernel bypass networking. Those last techniques have been shown to notably increase the scalability of media production on commodity servers.

Keywords — media processing, commodity servers, packet pacing, jitter, instrumentation, Field Programmable Gateway Array, kernel bypass, zero copy, software architecture

Résumé

La production de média pour la diffusion audiovisuelle (i.e., le processus par lequel plusieurs sources audiovisuelles sont mélangées et traitées en temps réel pour élaborer le flux consommé par le téléspectateur) est généralement implémentée par du matériel dédié, basé sur la Serial Digital Interface (SDI), une technologie d'interconnection dérivé de la télévision analogique. Malgré l'effort industriel présent pour remplacer le SDI par de l'IP (ainsi que spécifié par les standards SMPTE 2022-6 et 2110) la sensibilité au délai de la production de média rend difficile une transition totale vers un traitement logiciel sur des serveurs générique. Cette thèse résout different aspects de ce problème.

Premièrement, il a été conduit une étude quantitative et qualitative de la gigue subie par ces flux lors d'un traitement logiciel. Au delà de résultats obtenus pour des serveurs Linux x86_64, il a été dérivé une méthodologie générale, applicable à tout système d'exploitation et architecture matérielle, permettant d'étudier la gigue introduite.

Deuxièmement, une plateforme générique a été proposé afin de permettre la réalisation de système d'instrumentation personnalisé, pour l'horodatage précis de packet réseaux. Bien qu'étant basée sur la technologie des FPGA, cette plateforme permet à tout opérateur réseau ou de diffusion audiovisuelle de spécifier une logique d'horodatage personnalisée en utilisant le langage P4. Cela permet en particulier la conception d'une instrumentation pour la qualification de flux média.

Troisièmement, un système de lissage de traffic (packet-pacing) a été proposé, afin de permettre l'envoi de flux de paquets avec une gigue négligeable. Malgré un emploi exclusif de matériel générique, il a été prouvé formellement et expérimentalement que la gigue ainsi obtenue était suffisamment faible pour des flux média.

Finalement, un cadre logiciel facilitant l'écriture d'applications de traitement média a été proposé. Ce cadre repose sur la séparation entre le traitement et le transport des flux média, la couche de transport s'occupant du traitement haute performance des paquets réseaux par l'emploi de techniques comme le zero-copy, ou le kernel-bypass.

Mots-clefs — traitement média, lissage de trafic, gigue, instrumentation, serveurs standards, architecture logicielle, instrumentation, FPGA

Remerciements

Au nom de Dieu, le Tout Miséricordieux, le Très Miséricordieux

Avant tout, j'aimerais remercier mon cher directeur de thèse, Thomas Clausen, pour son dévouement et la qualité exceptionnelle de son encadrement. Thomas, je pense pouvoir dire sans me tromper, qu'il y a bien peu d'encadrants dont l'implication et le dévouement pourraient ne serait-ce qu'être comparés aux tiens. Bien au delà de ton rôle, tu m'as accompagné dans la découverte de la science des réseaux informatiques (en 2014 déjà), la construction de mon projet professionnel et académique, et tu m'as soutenu dans les moments les plus difficiles de ma thèse en ayant toujours un mot encourageant, sans pour autant compromettre les standards de qualité auxquels tu tiens des étudiants.

Bien évidemment, je ne peux parler d'encadrant de qualité sans remercier André Surcouf, qui est celui sans qui le fil conducteur même de ma thèse n'existerait pas. André, merci d'avoir proposé des problèmes techniques et scientifiques de qualité, d'avoir vu le potentiel de valorisation dans chacune de mes idées et de m'avoir exposé à la dure réalité de l'innovation dans le monde industriel, loin des rêves de laborantins. Je peux dire avec certitude que tu as contribué de façon décisive à construire l'ingénieur que je suis devenu.

Je remercie mes rapporteurs, Lars Eggert et Laurent Toutain pour leur lecture attentive de mon manuscript, ainsi que pour la qualité, la minutie, et la pertinence de leurs rapports. Merci également aux autres membres du jury, Nadia Boukhatem, et Colin Perkins pour leur présence, et la qualité de leurs interventions pendant la soutenance.

Merci également à mes collègues thésards et jeunes diplômés à Cisco : Aloÿs Augustin pour m'avoir prouvé qu'il n'y a pas besoin de faire un doctorat pour atteindre les 1000 citations, Yoann Desmouceaux pour son mentoring délicat et pour m'avoir montré que toute idée mérite exploration, Marcel Enguehard pour son mentoring incisif, et m'avoir montré que tout résultat mérite publication, Jacques Samain pour son expertise en chatbot déjantés, et Guillaume Ruty pour m'avoir appris à rester serein en toute circonstance.

Merci à Jordan Augé qui, par nos passionnantes discussions scientifiques et non scientifiques, a su rendre mes trajets quotidiens en RER C bien moins moroses. Merci également à Jérome Tollet, qui a su créer au sein de notre zone d'open space, une atmosphère équilibrée et agréable, teintée de professionnalisme, d'enthousiasme, de passion, mais aussi d'humour.

Merci à tout mes collègues de Cisco qui ont contribué techniquement et humainement à mon épanouissement durant ces trois années : Axel Taldir, Pierre Pfister, Mark Townsley, Julien Barbot, Fabien Andrieux, Franck

Bachet, Enzo Fenoglio, David Gaumont, Jeremie Garnier, Ghislain Bourgin, Alain Fiocco, Benoît Ganne, Guillaume Ladhuie, Malycia Ly, Guillaume Sauvage de Saint Marc, Hassen Siad, Éric Vyncke, Zhiyuan Yao, Thomas Feltin, Giovanna Carofiglio, Lura Muscariello, Alberto Compagno, Michele Papalini, Mohsin Kazmi, Neale Ranns, Nathan Skrzypczak, Arthur Toussaint et tant d'autres.

Merci à Jiazi Yi et Juan-Antonio Cordero-Fuertes, véritables piliers de l'équipe de recherche à l'école Polytechnique pour leur soutien et leur aide dans la rédaction d'articles. Je souhaite également remercier les membres de l'administration de l'école doctorale de l'École polytechnique pour leur incroyable flexibilité et leur accompagnement dans toutes les tracasseries administratives.

Des remerciements tout particuliers à Carole Reynaud, sans laquelle il m'aurait tout simplement été impossible de déjouer les subtilités administratives et/ou financières découlant de tout partenariat entre une entreprise d'envergure internationale et une entité académique publique.

Enfin je souhaite terminer par les remerciements les plus importants : ceux dédiés à mes parents ainsi qu'à mes frère et soeurs pour leur soutien au cours de cette aventure.

Contents

Chapter 1

Introduction

Prior to the development of general-purpose multi-core Central Processing Units (CPU) and commoditised, multi-gigabit Ethernet hardware, a class of performance-demanding applications was implemented either as Application-Specific Integrated Circuits (ASIC), or as software running on dedicated High-Performance Computing (HPC) clusters of — possibly application-specific — servers. These are interconnected with also proprietary interfaces such as InfiniBand [1] or Omni-Path [2]. The prohibitive Non-Recurring Engineering (NRE) costs of ASIC development are in opposition to the increasing demand for flexibility in application development. Moreover, despite ongoing standardisation efforts — *e.g.,* by the InfiniBand Trade Association [3] — HPC interconnection technologies are proprietary and subject to vendor lock-in, hence uncontrollable costs.

The availability of multi-core CPUs within the Intel x86_64 and ARM architecture, of Graphical Processing Units (GPU) bringing massive vectorised compute capacity, and of Ethernet networking equipment supporting rates above 10 Gbit/s, materialises a performance increase in general-purpose computing architectures. That initiated a paradigm change, even for the most specialised and performance-demanding applications: they are increasingly implemented modularly, as sets of **micro-services**, distributed across commodity servers in a data-centre. Such an implementation increases flexibility and vendor-independence, reduces the infrastructure cost, and enables **resource pooling**, which, ultimately, allows deploying applications **in the cloud** and full externalisation of all infrastructure-related costs.

This **transition from dedicated to commodity hardware** has benefitted fields such as scientific computing, machine learning and artificial intelligence [4–9]. Those workloads have the particularity of not being subject to tight timing constraints, *i.e.,* tasks do not have tight deadlines by which they must be completed. Therefore, that transition could be achieved

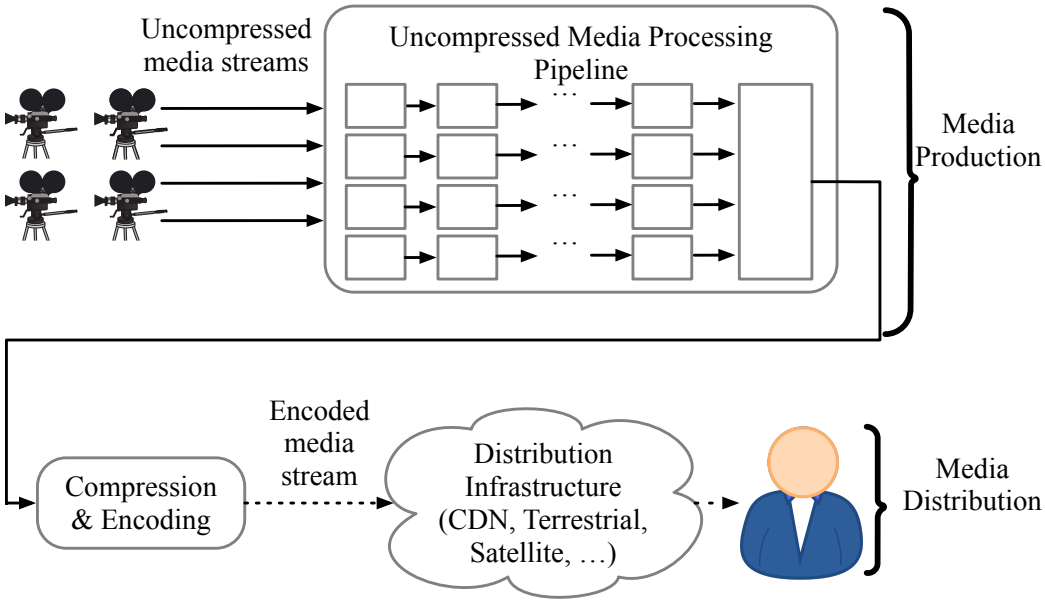


Figure 1.1: Functional overview of professional broadcasting

without considering the ability of the used commodity hardware, and of the software-stack (*e.g.,* operating systems, hypervisors, and application-level frameworks), to deliver guaranteed delays and task completion times.

In this thesis, transition from dedicated to commodity hardware is studied for a class of performance-demanding applications, subject to real-time constraints, rendering them more challenging to implement on commodity hardware: media-production for professional broadcasting.

1.1 Professional Broadcasting

Professional broadcasting is the activity of acquiring one or multiple media streams, applying a pipeline of media processing operations, and delivering the processed media stream to the end users. A functional, technology-independent, overview of a generic professional broadcasting setup is depicted in figure 1.1, where a set of media streams is generated by media sources, *e.g.,* video cameras, and microphones.

Media streams are subject to media processing, *i.e.,* to a sequence of audiovisual content alterations such as image insertion (*e.g.,* a television channel's logo), text insertion (*e.g.,* contextual banners related to the broadcasted program), video frame-rate adaptation, video deinterlacing, audio signal pro-

cessing, etc. Eventually, all those streams are *mixed*, *i.e.,* are merged into a single stream, which will transport the content as seen by the end-user. Mixing can consist of either *selecting* only one media input, or *merging* multiple streams according to various strategies (*e.g.,* picture-in-picture, mosaic, etc...). Those operations constitute the *media production* phase, as its finality is the production (*i.e.,* the elaboration) of the content as finally viewed by the end-user. Any audio or video compression occurring before the end of media production would result into quality degradation and latency increase — media streams are, therefore, kept uncompressed during media production.

The resulting stream is uncompressed and has a high data rate (*e.g.,* in the order of a gigabit-per-second for a High-Definition (HD) video stream at 30 Frames Per Second (FPS)), which makes it impractical for delivery to a client. This uncompressed stream is, therefore, to be prepared for delivery during a subsequent *media distribution* phase, starting with compression and encoding. Those result in an encoded stream whose data-rate is in the megabit-per-second order, depending on the used compression and encoding algorithms. Finally, the encoded stream is transmitted to the end-user over one or multiple distribution channels such as Terrestrial television or radio, Satellite links, Content Distribution Networks (CDN) for Internet distribution, etc.

As shown in table 1.1, media production and distribution have different requirements. For example, a delay in the order of seconds is acceptable between the beginning and the end of media distribution (*i.e.,* between the end of media production and media consumption by a client). However, a delay exceeding hundreds of milliseconds should not be introduced by media production, given that it includes some processing steps — *e.g.,* mixing — which are controlled by humans operators, who, therefore, need to visualise the output of media production. That *visual feedback* must be obtained with a latency imperceptible to a human operator, *i.e.,* in the order of hundreds of milliseconds at most.

As a consequence of those different requirements, the methods applicable to migrate media distribution from dedicated to commodity hardware are different from the ones applicable to migrate media production. Reviews of state-of-the-art solutions to implement media distribution and media production are given in sections 1.1.1 and 1.1.2, respectively. As detailed in the following, the media distribution field is more advanced in its migration towards commodity hardware than is the media production field, as the former is subject to less challenging data-rates and delay constraints than the latter.

	Media Distribution	Media Production
Data rates	≈ 1 Mbit/s (compressed audio and video)	≈ 1 Gbit/s (raw audio and videos)
Acceptable latency	Seconds	Hundreds of milliseconds
Stream replication	Thousands of clients for a given stream	Almost none (only a reduced number of endpoints involved in production)
Control over the infrastructure	Low because distribution depends on the medium used by the end-user (terrestrial, satellite, Internet, ...).	High because media production is performed in controlled studios.

Table 1.1: Requirements for media distribution vs production

1.1.1 Media Distribution: From Internet Protocol Television (IPTV) to Over-The-Top (OTT)

The development of Internet Protocol Television (IPTV) [10–12] demonstrated the feasibility of using commodity hardware and general-purpose networking equipment to implement media distribution by Internet Service Providers (ISP). IPTV relies on the ISP's Internet Protocol (IP) networks to deliver media content, transported by IP packets.

Because IPTV is implemented by ISPs, it can use two features which are only available across operator networks, *i.e.,* networks that are managed by a single operator, as opposed to features available across the wider Internet. First, IPTV leverages IP multicast (which is unavailable across the Internet), avoiding redundant packet transmissions and reducing the load on the operator network. Second, Quality-of-Service (QoS) policies can be consistently defined and enforced in operator networks, so as to prioritise IPTV streams [13], since those are less resilient to packet losses than, *e.g.,* TCP-based web traffic.

Over-The-Top (OTT)[1] media distribution [14] also transports media content over IP packets. Specifically, to define how media content is encapsu-

[1]Here, OTT media distribution is only considered in the context of live content, *i.e.,* in the context of professional broadcast. In the literature, OTT may also refer to techniques used to distribute Video on Demand (VoD), which, unlike live content, is not subject to delay constraints.

lated into network streams, OTT media distribution standardised transport methods such as Dynamic Adaptive Streaming over HTTP (DASH) [15] and HTTP Live Streaming (HLS) [16]. However, that *OTT traffic* is transmitted — as the name suggests — over the Internet, and is, therefore, likely to traverse multiple independent operator networks. As a consequence, neither IP multicast nor QoS policing are applicable to OTT media distribution, differentiating it from IPTV. OTT traffic is necessarily unicast and is subject to best-effort service. Therefore, IP multicast and QoS policing (used in IPTV) need both be replaced by alternative techniques, which are not dependent on the network infrastructure, and which are detailed in the following.

First, because those transport methods use the Hypertext Transfer Protocol (HTTP), they can be implemented by reusing components alreadyused for content distribution, such as web servers and Content Delivery Networks (CDN) [17–21]. The caching and geographical replication features offered by CDNs enable network load reduction when streaming to several end-users, and hence, are an alternative to IP multicast (used in IPTV), although they are independent from the network infrastructure, and are usable across the Internet.

Second, to maintain a guaranteed Quality of Experience (QoE) in the absence of QoS policing, OTT media distribution exploits *adaptive bitrate streaming* [22–24], *i.e.,* the receiver-initiated adaptation of the bitrate of the streamed media content to the network conditions (as evaluated by the media receiver). For example, DASH and HLS belong to a class of transport methods, implementing adaptive bitrate streaming, and called HTTP Adaptive Streaming (HAS). HAS-based transport methods all rely on *chunking*, *i.e.,* the discretisation of the media stream into atomic elements, called *encoded chunks*. Encoded chunks are sequentially generated by the "Compression and Encoding" stage of figure 1.1. Specifically, that stage discretises the input uncompressed media stream into a sequence of *uncompressed chunks* — each typically amounting to two to four seconds of media content for DASH and HLS. Then, each uncompressed chunk is compressed and encoded using multiple encoding rate, and hence, with multiple quality levels, yielding chunks of various sizes. The resulting *encoded chunks* are finally made available for OTT distribution, through web servers and CDNs. Upon media playback, the receiver continuously downloads encoded chunks corresponding to consecutive intervals of time, and decodes them to reproduce the original media content. For each interval of time, depending on the current connectivity conditions, the receiver chooses the bitrate (and, therefore, size) of the encoded chunk to download — resulting into an implementation of adaptive bitrate streaming.

As documented in [25], media chunking and HAS both increase the delay

between media production and media playback at the receiver. That increase is mainly due to two reasons. First, the generation of an encoded chunk of two seconds requires the availability of two seconds of uncompressed media prior to their compression, encoding, and delivery via web servers and/or CDNs. Therefore, the Compression and Encoding stage must accumulate at least two seconds of uncompressed media content before emitting the first encoded chunk, *i.e.,* that stage adds a delay necessarily superior to two seconds. Second, playback by the receiver requires media buffering, to accommodate for any network delay variation (and avoid any playback interruption). This buffering occurs with the granularity of a full encoded chunk, *i.e.,* it introduces a delay, which is an integer multiple of the duration of an encoded chunk, *e.g.,* two seconds. That coarse granularity is therefore likely to increase the playback delay (as observed by the receiver).

To summarise, while the techniques developed for OTT media **distribution** have been showed to enable the transition of media distribution to a general-purpose infrastructure (*e.g.,* IP packet networks, web servers, CDNs), those techniques necessarily introduce a delay superior to a second, rendering them impractical to use for media **production**.

1.1.2 Media Production: The Serial Digital Interface (SDI)

Media production has historically been implemented by chaining dedicated hardware appliances, interconnected by the Serial Digital Interface (SDI) [26–29], as standardised by the Society of Motion Picture and Television Engineers (SMPTE). SDI specifies an encapsulation of media content over coaxial cables, at a fixed data-rate of 270 Mbit/s for Standard Television Television (SDTV), and either 1.485 Gbit/s, or $\frac{1.485\,\text{Gbit/s}}{1.001} \approx 1.483\,\text{Gbit/s}$[2] for High-Definition Television (HDTV). SDI was designed as a replacement for analog video transmission standards [31] used in media production before the development of digital television [32–34]. As a consequence, SDI shares several features with analog video transmission standards over coaxial cables, and offers a type of connectivity that is fundamentally different from that provided by IP networking.

[2]The latter 1.483 Gbit/s rate is mainly used in North America, and the 1.001 ratio can be traced back to the frame rate of $\frac{30}{1.001} \approx 29.97$ FPS defined by the National Television System Committee (NTSC) analog color system [30].

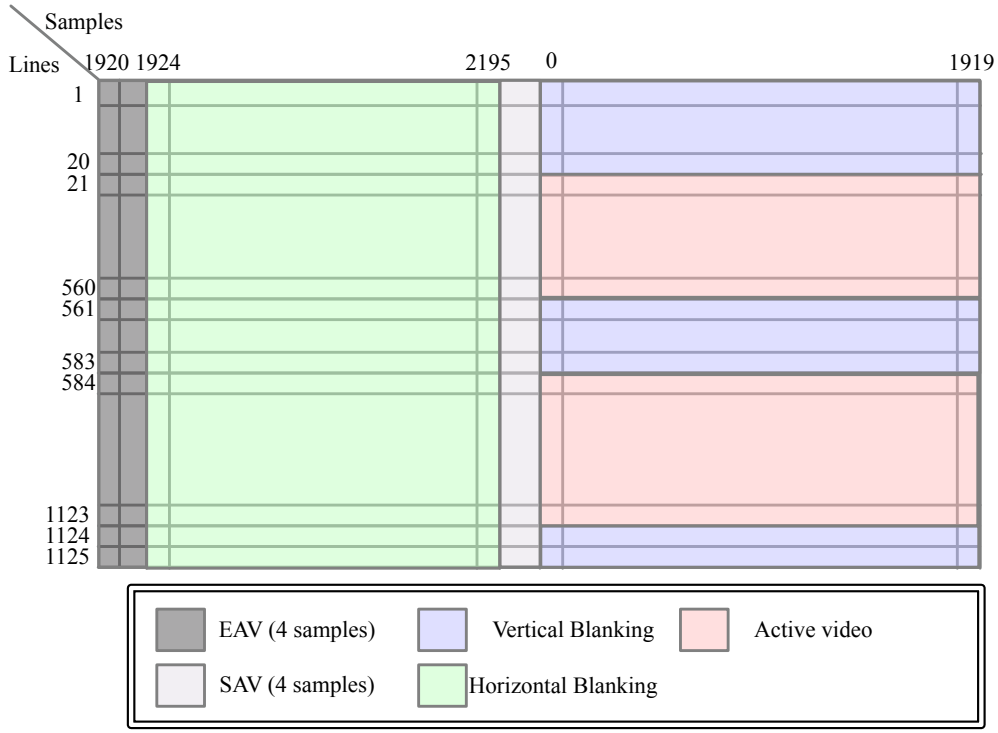


Figure 1.2: Structure of an SDI video frame encoding a High-Definition, interlaced, 1920x1080, 30 frames-per-second video stream. Each line contains synchronisation sequences denoted as End of Active Video (EAV) and Start of Active Video (SAV). Synchronisation sequences are 4-samples long and contain information describing the current line. The frame is separated in two fields from line 21 to 560 and 584 to 1123, corresponding to oddly and evenly numbered lines on the video.

The Serial Digital Interface: Legacy from Analog Video

Following the SDI specification, a media stream is transmitted as a sequence of **video frames**. Each video frame is transmitted as a sequence of **video lines**, and each video line is transmitted as a sequence of **video samples**. A video sample is encoded as two consecutive **10-bits words**. This specifies a sequence of 10-bits words which are channel-coded — by scrambled Non Return to Zero Inverted (NRZI) coding [35] — and serially transmitted over the wire.

The information contained in a video frame and a video line are better understood when considering analog video and Cathode Ray Tube (CRT) displays. Schematically, a CRT screen is periodically scanned, line-by-line, by an electron beam, whose intensity is modulated by the image to be displayed,

i.e., by the input analog video signal. At the end of the scan of each line, the electron beam must be repositioned at the beginning of the next line. This repositioning time is the **Horizontal Blanking Interval (HBI)**, and is included in the input analog video signal. Similarly, at the end of the last scan line, the electron beam must be repositioned at the top of the screen, during the **Vertical Blanking Interval (VBI)**. Finally, to improve the perceived frame-rate, instead of displaying, *e.g.,* 30 images per second, **interlaced video** works by first scanning all the odd-numbered lines, and then, after a first VBI, scanning all the even-numbered ones. That doubles the refresh-rate, at the cost of only refreshing half the image at each scan, *i.e.,* between each VBI.

SDI was directly derived from the concepts from analog video thus, a video frame transmitted by SDI also provisions one or two (if the transmitted video is interlaced) VBIs. The video lines corresponding to those are defined as **inactive lines** while the video lines containing information used to display an image are **active lines**. Similarly, in an SDI active video line, some video samples correspond to the HBI and are **inactive samples**, while others each contain two 10-bits words encoding the color of an image pixel, and are **active samples**.

Figure 1.2 depicts the structure of an SDI frame containing interlaced, High-Definition (HD) video at 30 frames per second. This structure shows that a significant fraction of the 1.485 Gbit/s serial data stream corresponds to HBIs and VBIs and does not carry any active video, *i.e.,* any information required to display the current picture. To avoid wasting that fraction of the channel's capacity, for each SDI video frame, VBIs and HBIs are used to carry **ancillary data** [36], *i.e.,* non-video information related to the SDI video frame.

Ancillary data embedded in the HBI is called Horizontal Ancillary Data (HANC) and consists of content that needs to be synchronised with the granularity of a video line. With the example depicted in figure 1.2, that granularity is $\frac{1}{1125*30\,\mathrm{Hz}} \approx 29.6\,\mu\mathrm{s}$. Because of this fine granularity, HANC data is typically used to embed audio samples [37, 38], accurately synchronised with the current video. Ancillary data embedded in the VBI is called Vertical Ancillary Data (VANC) and consists of content that needs to be synchronised with the granularity of the full SDI video frame, *e.g.,* per-frame closed-captions[3] [40, 41].

Finally, an SDI receiver consumes SDI video frames at a fixed frame-rate, defined consistently across a **media production** setup. To guarantee

[3]This is, in essence, no different from the way teletext is embedded in the VBI of analog video broadcast transmissions [39].

such a consistent **frequency**, a common time reference — *i.e.,* **a clock** — must be delivered to every SDI-based appliance. Furthermore, media streams originating from different media sources (*e.g.,* different video cameras) must be sampled with the same **phase**, *i.e.,* the acquisition of the first video sample of the first video line must be synchronised across all media sources. Indeed, an SDI-based appliance mixing input streams with unsynchronised phases can only generate a consistent SDI output stream by artificially delaying all input streams, except the one with the most advanced phase. This is suboptimal, as media production is very sensitive to end-to-end delay, as summarised in table 1.1.

To ensure phase synchronisation in a media production setup, all the SDI-based appliances receive a shared **external signal**, defining the times at which the sampling of SDI video frames should start. A possible implementation of that external signal is the **tri-level synchronisation signal**, which has standardised electrical specifications [31]. Furthermore, because that external signal defines the occurrence times of periodic events (*i.e.,* SDI video frame sampling), it is usable as a clock shared by all SDI-based appliances and, therefore, also enables frequency synchronisation.

To summarise, the SDI specification is highly influenced by analog video transmission, and is very similar to the straight-forward digitisation thereof. As a consequence, SDI-based transmissions have a **fixed data-rate**, independently from the information actually transmitted, *i.e.,* whether it is only solely video, or video and ancillary data such as audio and closed-captions. Because of that fixed-rate, an external signal is distributed across a media production setup to synchronise all SDI-based appliances.

SDI Video Routers

As depicted in figure 1.1, a media production workflow consists of a chain of media sources and processing elements. However, directly interconnecting them with physical SDI cables limits the flexibility of the physical installation: any workflow evolution requires time-consuming, and error-prone, physical rewiring of the media production setup. That lacks of flexibility motivated the use of **SDI video routers** to interconnect media sources and processing elements.

SDI video routers – also called **video switchers** – appeared early during the transition from analog to digital television [42]. An SDI video router is a piece of equipment with multiple SDI inputs and outputs, and is configurable to replicate each input on one or multiple outputs. In a media production setup, all the SDI outputs from the media sources and processing elements are connected to the inputs of an SDI video router, and all the

SDI inputs from the media processing elements are connected to its outputs. As a consequence, any media-processing pipeline can be implemented by reconfiguring the SDI video router, with no physical rewiring. This allows to dynamically repurpose the physical infrastructure to accommodate for multiple media production pipelines. It is, therefore, a first step into media processing resource pooling.

SDI connectivity vs the IP networking model

The serial, fixed-rate, synchronous, data transmission model, as defined by SDI, is different from the packet-based IP networking model:

1. SDI only allows **point-to-point unidirectional** data transmission, *i.e.,* a physical SDI connexion has only one input and one output. Consequently, any media stream replication must be performed by a specific piece of equipment (an SDI video router). This is in contrast to the IP networking model allows either **point-to-multipoint unidirectional** data transmission (through IP multicast), or **point-to-point unidirectional or bidirectional** data transmission (through IP unicast). The network infrastructure, therefore, provides stream replication through IP multicast[4].

2. The SDI specification allows **no multiplexing** over a given physical channel, *i.e.,* one coaxial cable can transport only one media stream. For example, an SDI connection capable of transporting an HDTV stream (*e.g.,* at 1.485 Gbit/s) cannot be reconfigured to carry multiple SDTV streams (each of 270 Mbit/s), despite the SDI physical layer being theoretically capable of doing so.

   That is different from the IP network model, as the latter is packet-based, and enables the **multiplexing of packets belonging to different logical flows** over a given network path. In other words, the IP network model allows network capacity pooling across different logical flows.

3. In a media production setup, all SDI-based data transmission are **synchronous with a centralised clock**. This raises operational challenges because the tri-level synchronisation signal must be explicitly

[4]Here, IP multicast is assumed to be a viable option for the transport of media production streams, because the packet switches constituting a potential IP-based media production setup are assumed to be all operated by a single entity. It would not be the case if a media production setup spanned over multiple network domains which are connected through the Internet.

distributed to all appliances. For example, the lengths of the cables used to distribute that signal have an impact on the signal transmission delay, and thus, might cause inconsistent synchronisation of those appliances. The IP networking model does not make any synchronisation assumptions on the underlying link-level layer, which can, therefore, implement **asynchronous** packet transmissions. For example, at the physical-layer, a 802.3 Ethernet [43] packet is transmitted synchronously with a clock, which is local to the transmitter. At the receiver, the Physical Medium Attachment (PMA) layer includes a Clock and Data Recovery (CDR) function which, for each incoming packet transmission, synchronises with the transmitter's clock, and recovers the transmitted packet. As a consequence, no explicit clock distribution is needed across an IP/Ethernet network, making the latter easier to manage than an SDI-based installation.

The IP networking model differs from SDI connectivity in a way similar to how packet-switched and circuit-switched communications differ. For example, SDI video routers and Plain Old Telephone Service (POTS) switches are essentially similar, as switching occurs on a per-call basis (in the case of telephone services) or on a per-programme basis (in the case of media production for professional broadcast). This is opposed to packet switching, which occurs on a per-packet basis, with no notion of established connection. In that sense, a parallel can be made between the migration of media production from dedicated to commodity hardware, and the evolution from circuit-switched to packet-switched networking.

1.2 Media Production on Commodity Hardware

This section gives an overview of the existing standards and technologies, which enable the migration of media production from SDI-based to IP-based media transport, and from SDI-based circuit-switching to IP-based packet-switching. Furthermore, the main limitations of those standards and technologies are presented.

1.2.1 Packetising SDI

To bring the benefits from the IP networking model (*i.e.,* point-to-multipoint communication, the ability to multiplex multiple media streams over a single physical connection, and the absence of required physical-level clock synchronisation) to media production, a natural idea consists of packetising the serial data stream defined by the SDI standards, and encapsulating the result

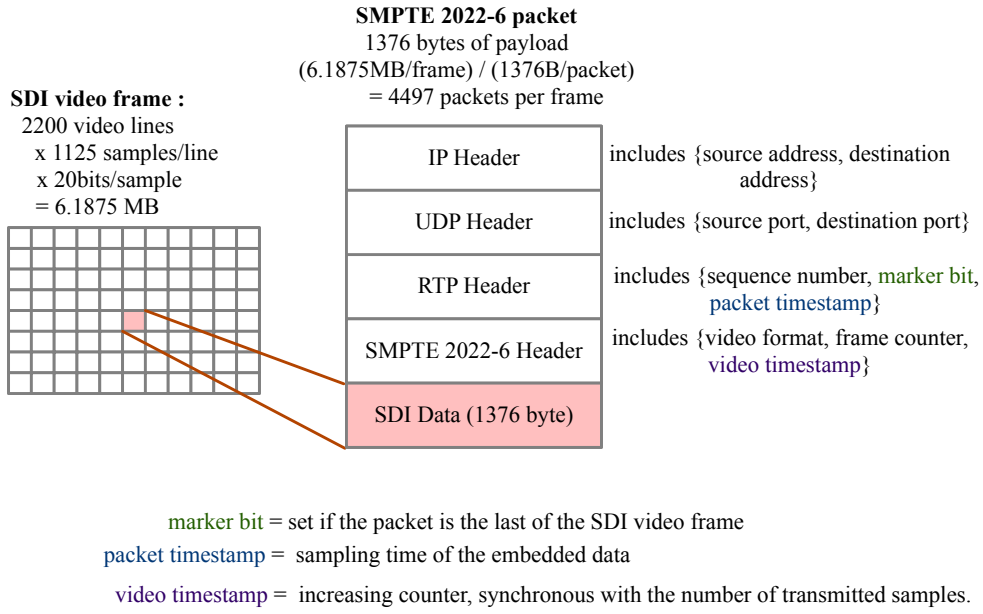


Figure 1.3: Structure of a SMPTE 2022-6 packet transporting an SDI stream

into IP packets. This approach was specified in 2012 by the SMPTE 2022-6 standard [44], and is historically the first step to enable the transport of media production streams over IP networks.

SMPTE 2022-6

As depicted in figure 1.3, a SMPTE 2022-6 packet includes IP, User Datagram Protocol (UDP) [45], Real-Time Protocol (RTP) [46] and SMPTE 2022-6 headers. The endpoint identifiers at the IP layer (source and destination addresses) and at the UDP layer (source and destination ports) provide a four-tuple, uniquely identifying a SMPTE 2022-6 media stream. As a consequence, packets belonging to different SMPTE 2022-6 media streams can be multiplexed over a shared commodity network infrastructure. Moreover, IP multicast enables in-network replication of IP packets, and hence, of media streams.

SMPTE 2022-6 packets are derived from SDI data by individually dividing each SDI video frame into a sequence of 1376-byte chunks. As the size of an SDI video frame is not necessarily a multiple of 1376 bytes, the last chunk of that sequence is completed with padding until its size reaches 1376 bytes. As depicted in figure 1.3, each chunk is prepended with a SMPTE 2022-6 and an RTP header.

The RTP header provides a packet sequence number and a packet timestamp. Because the IP network layer provides best-effort service, packet drops may occur. Those are detectable by the receiver, as a sequence number discontinuity. Moreover, best-effort service does not formally provide any guaranteed packet transmission delay, *i.e.,* the packet reception time is not sufficient to infer the associated media sampling time. It must therefore be embedded explicitly, as the RTP packet timestamp. This piece of data enables SMPTE 2022-6-based media processing to be performed synchronously with regards to a centralised clock, and hence enables mixing applications, similarly to the mixing of multiple, phase-synchronised, SDI streams. Finally, if the chunk transported by the RTP packet is the last of an SDI video frame, a **marker bit** contained in the RTP header is set.

The SMPTE 2022-6 header describes the format of the transported SDI stream (video frame-rate and resolution), and includes a frame counter and a video timestamp. The frame counter is incremented for each newly transmitted SDI frame, and thus provides a unique identifier for each of these. The video timestamp is a counter, giving the number of transmitted video samples (*i.e.,* pairs of 10-bit words) before the first video sample fully included in the current chunk. Contrary to the RTP packet timestamp, the value of the video timestamp is not necessarily synchronised with a centralised clock. As the frequency of the video timestamp is equal to the video sampling frequency (*e.g.,* for interlaced HD video at $\frac{30}{1.001}$FPS, that frequency is 1125 lines $\times$ 2200 samples $\times$ $\frac{30}{1.001}$FPS $= \frac{74.25}{1.001}$ MHz), it can be used by a SMPTE 2022-6 receiver to recover the underlying SDI video sample clock [44].

From the Tri-Level Synchronisation Signal to Precision Time Protocol (PTP) clocks

In an SDI-based installation, all equipment is synchronised by the tri-level synchronisation signal. Ethernet-based networks do not provide a similar physical-layer-based synchronisation feature. Thus SMPTE 2059 [47, 48] specifies the use of the Precision Time Protocol (PTP) [49] to distribute a common time reference in a media production setup. The Technical Recommendation 4 (TR-04) by the Video Service Forum (VSF) [50] completes the SMPTE-provided standards by specifying how the RTP packet timestamps from SMPTE 2022-6 are to be deterministically derived from a PTP-distributed time reference.

Use Case: Replacing SDI Video Routers with Packet Switches

Migrating a media production setup from dedicated to commodity hardware requires the replacement of SDI-based media-processing equipment with general-purpose, IP-based equipment. Because of the cost of this equipment, this migration is, most commonly, gradual . Specifically, the first step of this migration consists of only replacing SDI video routers with their IP-based counterparts, *i.e.,* IP/Ethernet packet switches transporting SMPTE 2022-6 streams [51]. This first step alone bring two operational benefits:

1. As the throughput of a single HD SMPTE 2022-6 stream is 1.5 Gbit/s, a single full-duplex 10 Gbit/s Ethernet port on a packet switch can simultaneously receive and transmit up to six different media streams. Therefore, it is functionally equivalent to six SDI inputs and six SDI outputs on an SDI video router.

   From an operational perspective, in a media production setup, transporting a given number of media streams requires less ports (and less cables) when media-transport is IP-based than when it is SDI-based.

2. Any physical piece of equipment implementing SDI-based video routing depends on the transported media format. For example, an SDI video router designed for SDTV streams cannot receive and transmit HDTV streams, despite the physical SDI connectors being identical. Consequently, any video format evolution requires upgrades of the SDI video routers interconnecting the media production setup. IP/Ethernet based packet switches are agnostic to the media format of the transported streams, provided that their throughputs do not exceed the network link capacities. That makes a media production setup based on IP/Ethernet packet switches more evolutive than one based on SDI video routers.

To enable a gradual migration from SDI-based to IP-based equipment, the replacement of SDI video routers with IP packet switches should be possible without replacing all SDI-based media processing appliances with IP-based ones. Therefore, to convert SDI streams into SMPTE 2022-6 ones (and vice versa), **SDI-to-IP (or IP-to-SDI) gateway devices** have appeared.

Specifically, media sources generate SDI streams, which are transformed into SMPTE 2022-6 streams and fed into network packet switches. For each step of a media processing pipeline, the SMPTE 2022-6 streams are switched to an IP-to-SDI gateway, media processing is performed by SDI-based equipment, and the resulting stream is transmitted to an SDI-to-IP gateway, and then back to the network packet switch. Such a **hybrid media production**

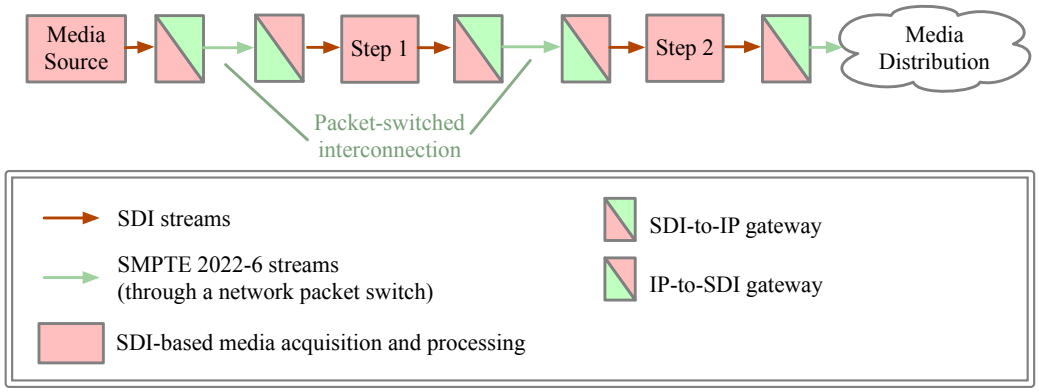


Figure 1.4: Functional illustration of an hybrid SDI and SMPTE 2022-6 media production setup. SDI-based media acquisition and processing components are interconnected by a packet switched architecture, SDI-to-IP and IP-to-SDI gateways.

setup, mixing SMPTE 2022-6 based interconnection (with commodity network packet switches) and SDI-based media acquisition and processing (with dedicated hardware) is depicted in figure 1.4.

While the sole replacement of SDI video routers with network packet switches brings clear operational benefits, it is only the first step in the migration from dedicated hardware to commodity hardware. The next natural step consists of directly processing SMPTE 2022-6, with commodity hardware capable of transmitting and receiving network packets.

1.2.2 Software-based Media Processing on Commodity Servers: Challenges and Limitations of SMPTE 2022-6

Processing a packet-based, SMPTE 2022-6 stream with commodity servers appears to be a straight-forward process, as those are equipped with inexpensive general-purpose Network Interface Cards (NIC), capable of receiving and transmitting network packets at multi-gigabit data-rates. However, while SMPTE 2022-6 enables replacing SDI-based connectivity in a hybrid media production setup, that standard is ill-adapted to software-based media processing for two main reasons: media essence multiplexing, and undefined packet transmission timing.

From Multiplexed To Separate Media Essences

As a packetised version of SDI, a SMPTE 2022-6 stream embeds audio, video, and ancillary data. When the receiver is implemented as SDI-based dedicated hardware, multiplexing those three logical data streams over a single serial digital transmission brings operational benefits, as it avoids the necessity for three distinct physical connexions. Moreover, the digital logic required to demultiplex video, audio and ancillary data has limited complexity, as the layout of an SDI video frame and the locations of VANC and HANC are well-defined by the SDI standards. In other words, multiplexing the audio, video, and ancillary **media essences**[5] in the SDI payload is adapted to media production setups where, eventually, media streams are received and processed as SDI streams — *e.g.,* the hybrid media production setup of figure 1.4.

However, if the receiver consists of software processing packets through a network stack, the previous benefits of multiplexed media essences vanish. First, a network path can already, natively, multiplex as many logical data streams as needed, each of which being identified by a unique **five-tuple** (IP protocol number, the source and destination IP addresses, and source and destination transport-layer ports). Five-tuple-based demultiplexing only requires parsing IP and UDP headers. Demultiplexing audio, video, and ancillary data streams from a SMPTE 2022-6 stream is more CPU-intensive, as it requires parsing each 10-bit word of each SMPTE 2022-6 packet. Further, in a software-based media production setup with multiple servers hosting media-processing application, audio, video and ancillary data streams are not necessarily processed at the same location, which makes the use of the multiplexed SMPTE 2022-6 format impractical.

Because multiplexed media essences — as required by SMPTE 2022-6 — are unadapted to software-based media processing, the SMPTE 2110 family of standards [52] was developed. It specifies distinct packet-based transport for each media essence: SMPTE 2110-20 [53] for video transport, SMPTE 2110-30 [54] for audio transport and SMPTE 2110-40 [55] for ancillary data transport.

While audio and ancillary data transported in an SDI or a SMPTE 2022-6 stream are naturally synchronised with the corresponding video (as they are associated with a given SDI video frame for VANC or with a given video line for HANC), that synchronisation disappears when decoupling the trans-

[5]Despite its lacking of a formal definition, the term of art **media essence** designates a sub-stream belonging to a media stream, and transporting media data of a single nature. Typically, a media stream has three media essences: video data, audio data, and ancillary data.

port of the different media essences, as specified by SMPTE 2110. SMPTE 2110-10 [56] addresses that issue by specifying a unified timing framework, connecting the different timestamps appearing in the packet transporting audio, video, and ancillary data.

To summarise, SMPTE 2110 enables separate processing of the audio, video and ancillary data component included in a media stream, without sacrificing the synchronisation between those components. That eases software-based processing, as it removes the requirement for any CPU-intensive demultiplexing of the different media essences included in an SDI or SMPTE 2022-6 stream. Moreover, it eases the pooling of compute and network resources in a data-centre, as the different essences are transported independently and can be processed by different pieces of equipment.

Undefined Packet Transmission Timing

The SMPTE 2022-6 standard only specifies a packet format to transport SDI data and does not mandate any property relating to packet transmission timing. As a consequence, it is implied that packet transmissions are lossless and occur in such a way that a SMPTE 2022-6 receiver is always able to consume a packet from its internal buffer, if it needs to do so. In other words, SMPTE 2022-6 does not include any flow or congestion control mechanism, but relies on the assumption that the network provides reliable transport and ensures transmission delays consistent with the packet consumption schedule of the receiver.

In practice, despite an unspecified timing behaviour, SMPTE 2022-6 transmitters are implemented so that packets are transmitted at a constant packet-rate, as implied, *e.g.,* by the data sheet of a Xilinx FPGA intellectual property core implementing an SDI to SMPTE 2022-6 converter [57]. However, even if packet transmissions occurs at a constant-rate, *i.e.,* with a constant Packet Inter-arrival Time (PIT), an IP/Ethernet packet-switched network introduces a non-constant delay, causing packet arrival jitter at a SMPTE 2022-6 receiver, which, therefore, needs buffer packets, so as to be able to consume them at a constant rate. Because SMPTE 2022-6 does not constraint that jitter, buffer dimensioning at the receiver is challenging and must be done empirically.

If SMPTE 2022-6 is only used to replace SDI video routers with packet switches in a hybrid media production setup as depicted in figure 1.4, packets are periodically transmitted by SDI-to-IP gateways, and periodically consumed by IP-to-SDI gateways. Latency measurements [58] show that packet switches typically used for hybrid media production introduce a jitter lower than two microseconds. As a consequence, and given that the nominal PIT

of a SMPTE 2022-6 stream — *e.g.,* 7.419 µs when transporting interlaced, 1920x1080, 30 frames-per-second video — is considerably higher than that jitter, very limited buffering at the IP-to-SDI gateway is sufficient to ensure that packet can be consumed at a constant-rate.

Conversely, commodity servers running packet-processing software introduce substantial jitter when receiving and transmitting SMPTE 2022-6 streams. Because SMPTE 2022-6 does not specify the jitter that a receiver must tolerate, it is impossible to formally specify real-time constraints on a packet-processing software, so that it accommodates any SMPTE 2022-6 receiver.

Consequently, the SMPTE 2110 family of standards constrains timing more explicitly than does SMPTE 2022-6. In particular, SMPTE 2110-21 [59] defines a set of packet transmission **profiles** to which any SMPTE 2110 sender must conform. A profile is formally defined by the behaviour of a packet receiver, *i.e.,* a receive buffer size, and a packet consumption schedule. A SMPTE 2110 sender conforms to a profile if no receive buffer overflow or starvation occurs for a hypothetical packet receiver behaving as defined in the profile, and connected to the sender by way of an hypothetical network introducing no delay.

For example, hardware-based receivers may have reduced buffering capabilities — *e.g.,* because they may be implemented on Field Programmable Gate Arrays with limited memory. Consequently, SMPTE 2110-21 defines a Narrow (N) profile, associated with a reduced buffer size. An SMPTE 2110 sender, conforming to the N profile, must transmit packets in such a way that a receiver with reduced buffering capabilities will experience no starvation nor overflow. In particular, such a sender is compatible with hardware-based receivers. Similarly, software-based receivers have higher buffering capabilities — *e.g.,* because modern NICs support large receive queue sizes and thus, SMPTE 2110-21 defines a Wide (W) profile, associated with a high buffer size. Consequently, senders compliant with the W profile are compatible with software-based receivers, but may incur buffer overflow or starvation on hardware-based receivers.

In summary, as neither any flow control method nor any packet transmission timing property are defined by SMPTE 2022-6, that standard cannot be used beyond the scope of hardware-based SMPTE 2022-6 generation (through SDI-to-IP gateways), delivery (through packet switches) and consumption (through IP-to-SDI gateways), *i.e.,* it cannot be used for software-based media production stream processing. Indeed, the latter yields non-negligible jitter thus, an accurate specification of packet transmission timing is needed, which is achieved by the SMPTE 2110-21 standard. That standard specifies the packet transmission regularity required from a SMPTE 2110

sender by providing normative information, which is differentiated depending on the type of the targeted receiver (software- or hardware-based).

Migrating media production from dedicated hardware to commodity servers and networking equipment is a gradual process, and raises challenges due to fundamental differences between SDI-based and packet-switched data transport. While the SMPTE 2022-6 and 2110 standards have been shown to offer the tools to accurately **specify** the desired behaviour of both hardware-based or software-based packet-based media production components, those standards leave open the problem of **implementing** such components.

1.3 Thesis Contributions

Going one step beyond the **specification problem** (largely covered by SMPTE 2110 and SMPTE 2022-6), this thesis addresses the **implementation problem**, which is:

> **To what extent can software-based media production be realised on commodity servers and general-purpose networking hardware?**

Chapter 2 proposes a reproducible, software-based, experimental methodology to evaluate the jitter incurred by a periodic packet stream — such as a SMPTE 2022-6 stream — upon reception by software executed on a commodity server. Specifically, that methodology includes the exhaustive enumeration of all jitter sources — depending on the hardware, the used operating system, the properties of the network-stack, etc — and the relative contributions thereof to the overall jitter. That methodology is applied to a Linux-based, x86_64 commodity server receiving a SMPTE 2022-6 stream. The obtained experimental results support the fact that, as stated in section 1.2, software-based packet processing for media production introduces an amount of jitter which is incompatible with hardware-based receivers with a limited buffering capability.

The contribution presented in chapter 3 extends jitter evaluation to hardware-based methods. Specifically, the development of hardware-based packet timestamping tools is made easier, through the proposal of a novel FPGA-based framework: the Open Platform For Programmable Precise Packet Timestamping (OP4T). By leveraging the packet-processing-oriented P4 programming language [60], OP4T renders the design of custom, hardware-based measurement tools, accessible to broadcasting and network operators with little hardware design expertise. Furthermore, an implementation of OP4T is experimentally evaluated, and shown to allow packet timestamping with a

microsecond-scale accuracy — which is necessary to assess the conformity of media production streams to packet timing profiles, such as those specified in SMPTE 2110-21.

Chapter 4 offers an extensive analysis of packet pacing on commodity server, *i.e.,* the transmission of a packet stream, as periodically as possible, and with a minimal jitter. To that end, a mathematical definition of jitter is first proposed, and then, is used to expose the root causes limiting the accuracy of software-based packet pacing on commodity servers. To overcome that limitation, this chapter introduces the notion of **pacing assistant**, *i.e.,* an auxiliary component, external to the server, and which is able to provide sufficient assistance to the server, to enable high-accuracy packet pacing. A pacing architecture, and algorithms relying on a pacing assistant, are given and the achievable jitter is mathematically quantified. Furthermore, pacing assistants are shown to be effectively implementable with existing commodity hardware and packet switches and thus, are proven not to be purely abstract constructs. In summary, the work of chapter 4 is a key enabler for the transmission of low-jitter SMPTE 2022-6 and 2110 packet streams for media production on commodity servers.

Finally, chapter 5 proposes a software framework and architecture, which ease the implementation of media-processing applications. That is achieved by abstracting media transport away from media processing, *i.e.,* by hiding the entire packet processing details from media-processing applications. Therefore, instead of the traditional socket API, a media-processing application is exposed a **virtual Media Interface (vMI)**, which, instead of packets, relies on a coarser data unit: **media frames**. A media frame is a generalisation of an SDI video frame, *i.e.,* correspond to a an atomic element belonging to a media stream. In chapter 5, that vMI architecture is specified, implemented, and shown to improve the performance and scalability of realistic media processing pipelines.

1.4 Publications and Software Production

This section enumerates the scientific and software production resulting from this thesis.

- Arthur Toussaint, Mohammed Hawari, Thomas Clausen, "Chasing Linux Jitter Sources for Uncompressed Video" in *2018 14th International Conference on Network and Service Management (CNSM)*, derived from the work presented in chapter 2.

- Mohammed Hawari, Juan-Antonio Cordero-Fuertes, Thomas Clausen,

"High-Accuracy Packet Pacing on Commodity Servers for Constant-Rate Flows" accepted for publication in *IEEE/ACM Transactions On Networking*, derived from the work presented in chapter 4.

- Mohammed Hawari, Axel Taldir, André Surcouf, Yoann Desmouceaux, Thomas Clausen, "vMI: Software Architecture for Transparent High-Performance Media Transport" submission in preparation, derived from the work presented in chapter 5.

- Mohammed Hawari, Thomas Clausen, "OP4T: Bringing Advanced Network Packet Timestamping into the Field" submission in preparation, derived from the work presented in chapter 3.

- The software and hardware design developed for OP4T, open-source release in preparation, derived from the work presented in chapter 3.

- Cisco Herisson, available at `https://github.com/cisco/herisson` as open source code, derived from the work of chapters 4 and 5.

Chapter 2

Chasing Linux Jitter Sources for Uncompressed Video

As discussed in chapter 1, the development of network-based transport for media streams — such as SMPTE 2022-6 and SMPTE 2110 — is an enabler for software-based media processing on commodity servers, because those have Network Interface Cards (NICs), and, therefore, can receive, process, and transmit network packets. However, those streams have high data-rates — around 1.5 Gbit/s for SMPTE 2022-6 — and high packet-rates — around 135000 packets per second for SMPTE 2022-6 — causing Packet Inter-arrival Times (PIT) in the order of a 7.41 µs. Moreover, chapter 1 detailed the reasons justifying those streams undergoing Constant Rate (CR) packet transmission, and justifying their sensitivity to packet jitter. As a consequence, it is critical to understand and to be able to quantify the jitter introduced by software-based packet processing of CR packet streams, when it is performed on commodity servers running general-purpose Operating Systems (OS). An understanding of this jitter informs of the suitability of using commodity servers for software-based media-processing, and informs on the buffering capacity required at a SMPTE 2022-6 or 2110 receiver consuming a software-processed media stream.

In this chapter, the term Video Processing Function (VPF) generically designates a piece of software receiving a packet-based media stream.

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 [61] – 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 [62, 63].

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, [64] 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 [65] of periodic networked systems with events in the order of 100 µs on a FreeBSD-based Commercial Off-The-Shelf (COTS) server, little attention has been given to characterising jitter on periodic events.

Yet, with SMPTE 2022-6 receivers expecting a CR stream giving rise to a packet arrival time with a periodicity in the order of 7.41 µs, if a VPF is to be successfully executed on a COTS server, a granular understanding of its jitter properties is required.

Statement of Purpose

This chapter characterises the jitter, introduced by a COTS x86 server running a Linux-based operating system, upon **reception** of network packets corresponding to a SMPTE 2022-6 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.

Chapter Outline

The remainder of this chapter is organised as follows: section 2.1 describes the data-path taken by a packet, *from wire to application*, enumerating the potential sources of jitter that can be encountered. Section 2.2 motivates and introduces the experimental setup used to quantify these sources of jitter, which is then used for producing the results presented in section 2.3. This chapter is concluded in section 2.4.

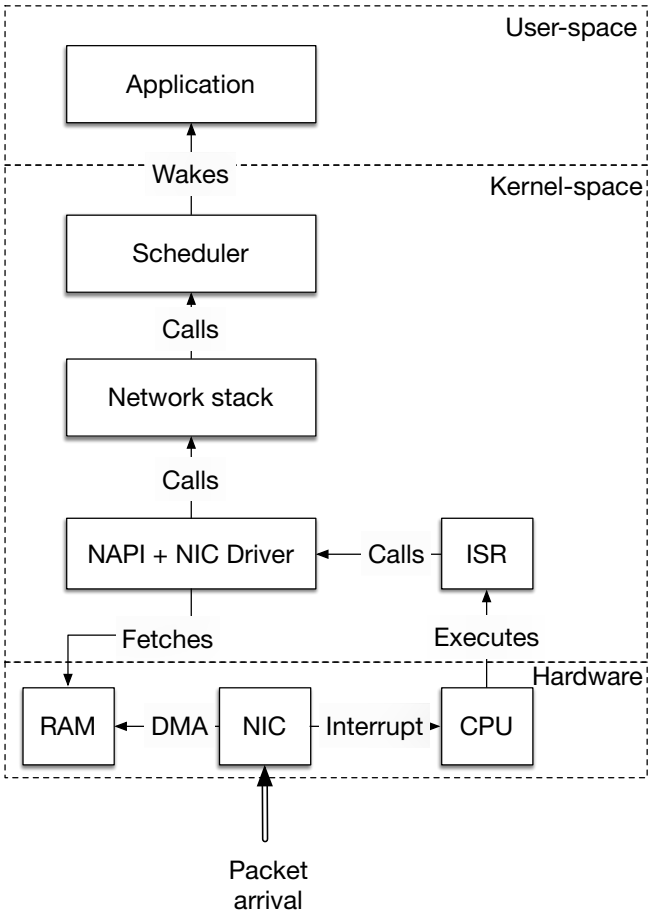


Figure 2.1: Schematic view of the path taken by a data packet – from Network Interface Card (NIC) to Application.

2.1 From wire to application

The jitter sources along the path of a packet through a COTS server, depicted in figure 2.1, from its arrival at the Network Interface Card (NIC) until it is delivered to an application, are analysed in this section.

2.1.1 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.

2.1.2 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 2.1, 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.

2.1.3 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 2.1.2 and illustrated in figure 2.1.

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.

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.

2.2 Experimental Setup

To quantify the contribution of the potential jitter sources, identified in section 2.1, to the overall jitter of a VPF receiving a SMPTE 2022-6 stream, each is studied in an isolated environment.

2.2.1 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.

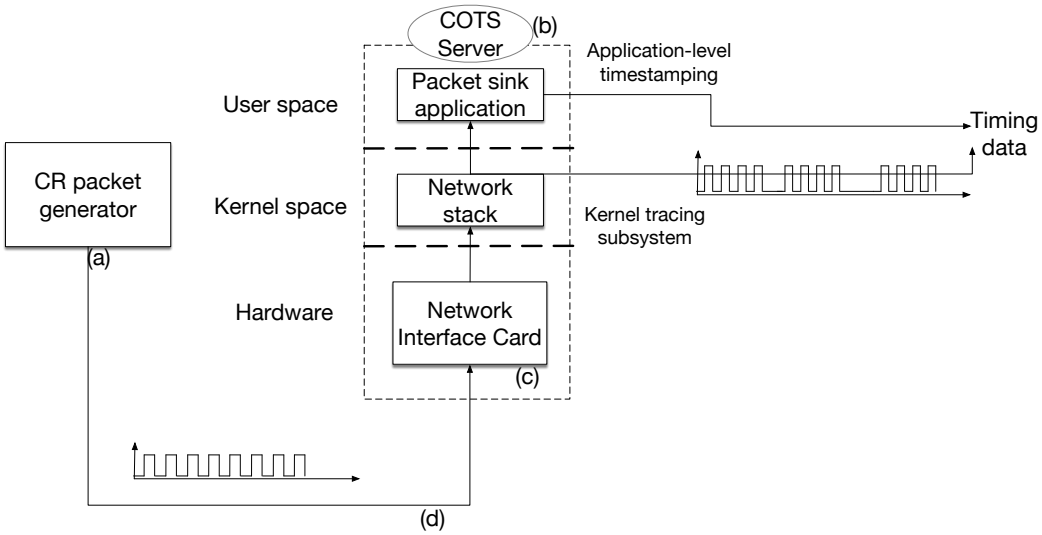


Figure 2.2: Abstract view of the analysed system

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.

2.2.2 Quantitative Scope

The test setup is illustrated in figure 2.2, 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 2.2 was substituted by a SMPTE 2022-6 hardware network analyser[2]. The measurement results are depicted in figure 2.3 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, CR with a precision of 1 µs, and any sub-microsecond packet delay variation observed, therefore, cannot be attributed to the server hardware or software in this setup.

[1] https://www.kernel.org/doc/html/v4.17/trace/index.html
[2] Specifically, a Tektronix PRISM.

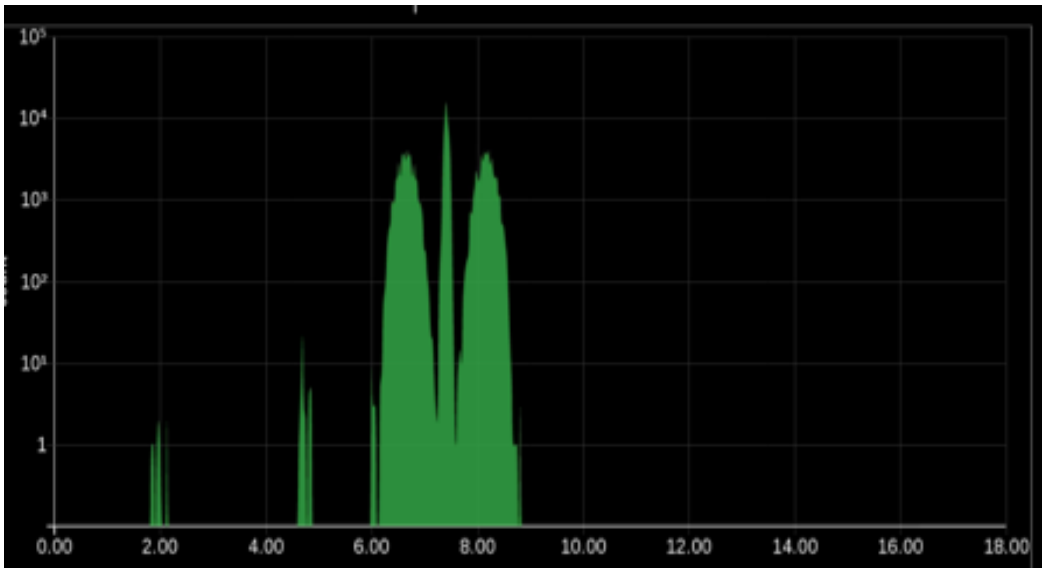


Figure 2.3: Histogram of the packet inter-arrival times at the ingress of the server

2.2.3 Hardware setup

The hardware setup is as follows, with reference to figure 2.2:

(a) A commercial SDI to SMPTE 2022-6 converter configured to output a SMPTE 2022-6 stream encapsulating a 1080i 29.97 frames per second video test pattern is used as **CR 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 Gbit/s optical interface is used as the **Ingress NIC**.

(d) The interconnection between the packet generator and the COTS server is implemented by a Cisco Nexus 9000 fully non-blocking switch.

2.3 Experiments and Results

This section experimentally quantifies the contribution of each source of jitter, as enumerated in section 2.1. In the setup of section 2.2, all known sources of jitter eliminated, a baseline is established. These sources are then restored one by one and their impact is measured.

Table 2.1: Available kernel options to reduce network-independent jitter

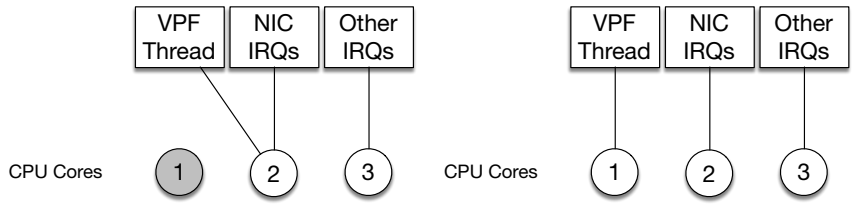
Kernel Option	Description
`nohz_full` = <CPU_LIST>	For each CPU core in <CPU_LIST>, disables the scheduler periodic tick when at most one thread is runnable on it.
`isolcpus` = <CPU_LIST>	Prevents the CPU cores in <CPU_LIST> from running any threads that were not explicitly assigned to a core in the 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.

2.3.1 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, handling NIC interrupts, and handling other (non-NIC) interrupts. This is done by way of using the Linux kernel options, indicated in table 2.1, as follows:

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

Moreover, as illustrated in figure 2.4a, the VPF is shielded from all interrupts



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

Figure 2.4: Interrupt and scheduling affinity. Involved cores are identified as core 1, 2 and 3.

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 figure2.4a, 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 2.1):

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

With the same objective, 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 2.4a, 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 2.2 is implemented in polling mode; by calling `recvfrom` in a tight loop with the `MSG_DONTWAIT` flag, the VPF never blocks which eliminates the call to the scheduler evoked in section 2.1.

2.3.2 Baseline: Experiments and Results

To differentiate between network-independent jitter, which was defined in section 2.1.3, and the jitter introduced by the processing of incoming packets, a *dummy program* is implemented; it consists of a loop, busy waiting for 7.41 µs 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 µs 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 2.5 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 µs 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 2.6b depicts the time series of packet inter-arrival times as measured by the VPF, while figure 2.6a 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 2.5 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 2.5 and figure 2.6b, and (ii) network jitter as shown by the similarity of the 1 µs-wide noise around the 7.41 µs average in figure 2.6a and 2.6b.

Figure 2.7a and figure 2.7b 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 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 2.7b 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

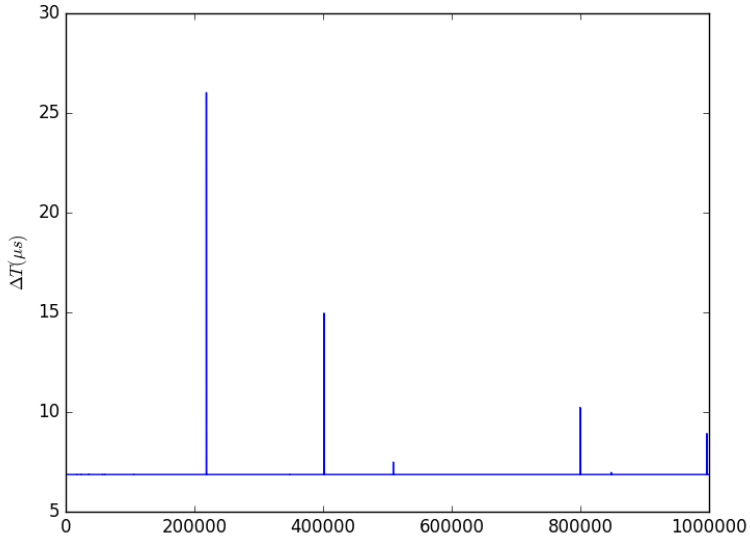


Figure 2.5: Dummy program : 1M acquisitions

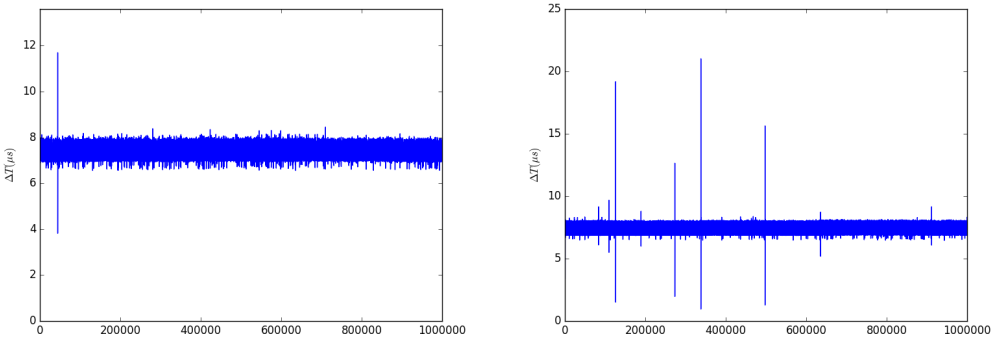
enumerated in section 2.1 are restored independently so as to study their relative contribution.

2.3.3 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 2.8 shows the apparition of many spikes in the order of 15 µs when compared to figure 2.6a.

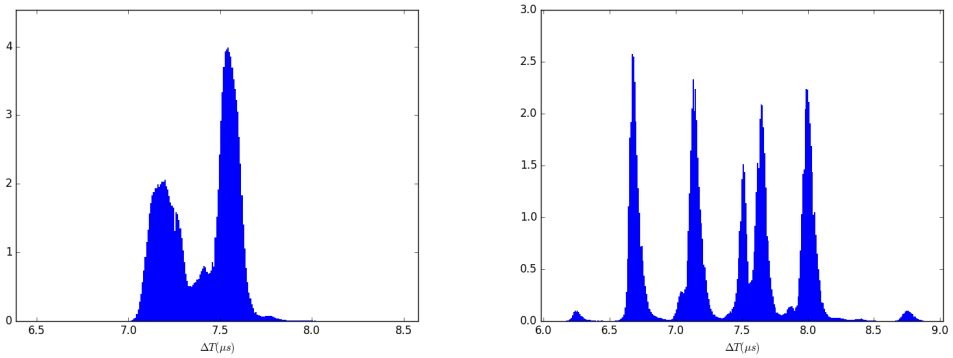
When interrupts are routed to a different CPU core than the one on which the VPF is scheduled, it is plausible to assume jitter reduction as ISRs are granted a dedicated core. Jitter increase is also plausible because of the requirement for inter-core synchronisation, which is a source of jitter *i.e.,* 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 2.6a) NIC interrupts are routed to the same CPU core as the one on which the VPF is scheduled, in the second (figure 2.9a), NIC interrupts are *re-routed* to a different CPU



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

Figure 2.6: Baseline system: time series



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

Figure 2.7: Baseline system: Histogram of ΔT

core, and on the third (figure 2.9b), 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 2.4b. 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).

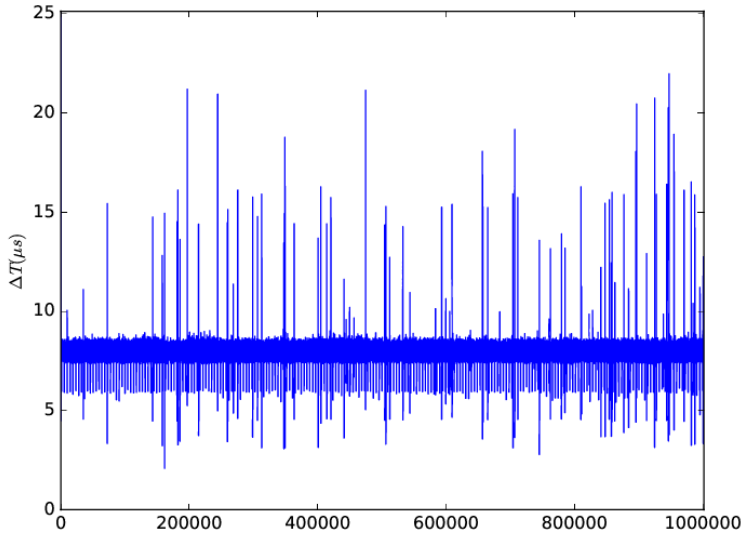


Figure 2.8: System without `idle=poll`: tracing `irq_entry`

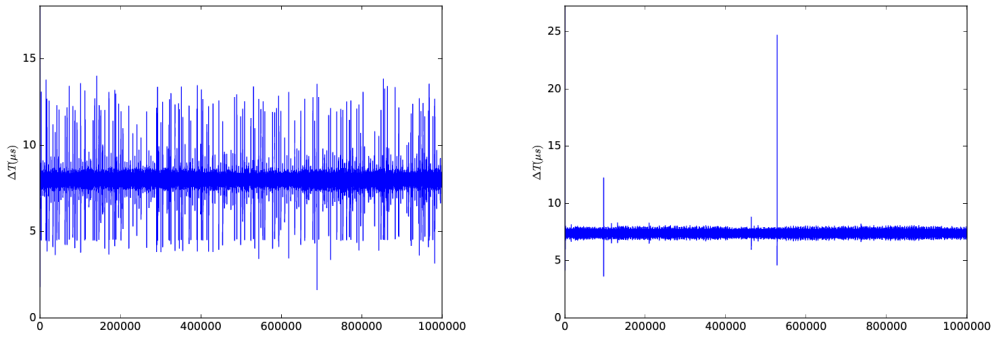
2.3.4 Linux Scheduler induced jitter

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 2.10. Without the latter kernel option, this blocking leads to even more jitter, showing spikes of up to 60μs

As explained in section 2.3.1, that jitter can be eliminated by receiving packets in polling mode.

2.3.5 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



(a) With infinite loop

(b) Without infinite loop

Figure 2.9: System with re-routed interrupts: `irq_entry`

elapsed.

As described in [66], 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 2.11.

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 2.11. Therefore, ITR hides all the other sources of latency studied here.

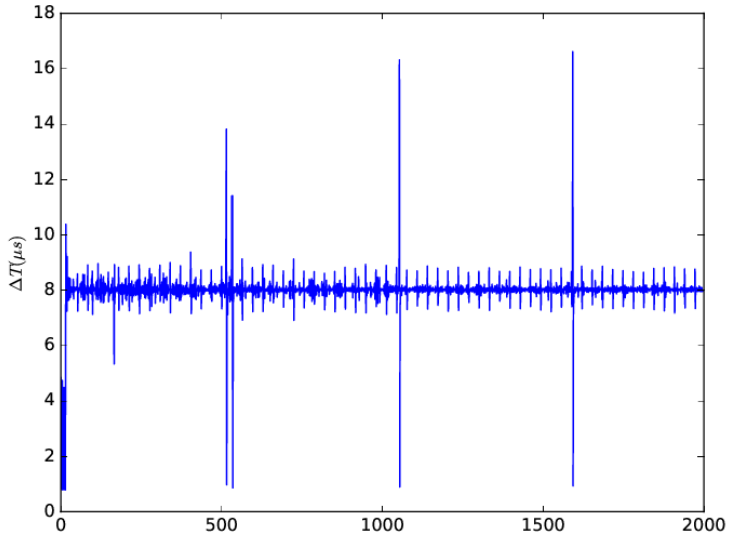


Figure 2.10: Measurements in blocking mode: VPF

2.4 Conclusion

As media 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, media 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 SMPTE 2022-6 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 chapter 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 SMPTE 2022-6 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

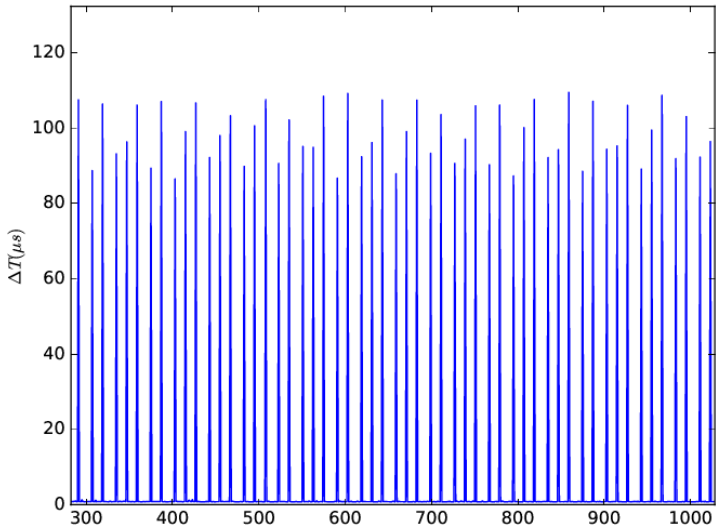


Figure 2.11: Measurements with ITR activated and setting of $T_{ITR} = 100\mu$s

implementation of software-based VPFs in a variety of environments.

Chapter 3

OP4T: Bringing Advanced Network Packet Timestamping into the Field

While the methodology developed in chapter 2 allows characterising the jitter incurred on a constant-rate media stream when it is received by a piece of software, the implemented technique relies on *software timestamping*. As a consequence, the accuracy of the obtained timestamps is limited by some of the identified jitter sources. For example, the network-independent jitter sources evoked in the conclusion of chapter 2 — such as the occurrence of unpredictable and uncontrollable System Management Interrupts — severely limit the ability to perform software-based packet timestamping. This chapter proposes a generic hardware platform, allowing to build hardware-based instrumentation to perform high-accuracy and flexible packet timestamping.

As detailed in the following, the latency evaluation problem addressed in chapter 3 has a scope beyond media production, and spans over the design and management of any latency-sensitive networked system.

Background

Accurate measurement of latency is a key tool for qualifying the performance of networked systems. Previous work has shown that traffic patterns in data-centre networks include packet bursts [67, 68], observed as a heavy-tailed distribution of packet interarrival times [69]. Bursts are responsible for an increased buffer occupation in packet switches, eventually leading to queuing and, if buffers are undersized, packet drops. The corresponding additional delays, even when they are in the microsecond scale, are in turn re-

sponsible for observable, application-level, performance impairments [68, 70]. Moreover, packet bursts in data-centre networks are transient, appear at time-scales in the order of a few dozen micro-seconds [71], and are difficult to detect by coarse measurements. That is different from traffic patterns, occurring in wide-area networks, and observable by methods such as tomographic inference [72], which are derived from coarse metrics,

To understand such transient network traffic patterns, and to diagnose transient latency spikes, instrumentation enabling accurate packet timestamping on selected flows is, therefore, crucial. Such instrumentation is already available as a part of *network testers*, *i.e.,* systems capable of generating predefined traffic patterns, and monitoring the latency introduced by networked Devices Under Test (DUTs). Despite the prior existence of network testers, both as commercial hardware appliances — *e.g.,* Ixia PerfectStorm or Spirent TestCenter, and as open source hardware designs — *e.g.,* the Open Source Network Tester (OSNT) [73] or FlueNT10G [74], the cost, programmability and/or performance of those solutions are subject to limitations, described in this chapter. More fundamentally, those network testers are only designed to be used during the qualification phase of a DUT, and not *in situ*, *i.e.,* for understanding latency issues in a real deployment.

Statement of Purpose

This chapter extends beyond the scope of simple network testers by introducing the Open Platform for Programmable Precise Packet Timestamping (OP4T). While network testers are external to a DUT, and are responsible both for generating traffic patterns and for monitoring a temporal response, OP4T exposes a deliberately different semantic; OP4T belongs to the category of Smart Network Interface Card (SmartNIC) and exposes the same services as a regular network interface. Used in a data-centre server in place of a commodity Network Interface Card (NIC), OP4T enables in-band packet timestamping, with a minimal disruption of normal application operations.

The OP4T architecture is designed according to five guiding principles.

1. **Openness** Primarily targeted towards the research community, OP4T must be compatible with an affordable network prototyping Field-Programmable Gate Array (FPGA) board and, as much as possible, must reuse existing open-source hardware designs.

2. **Programmability** OP4T must allow programmable packet timestamping and payload alteration, to enable selecting the packet flows to monitor, and, potentially, those to alter with in-band timestamps. As such

programmability must be accessible to network operators, not necessarily specialised in field programmable logic design, OP4T must provide a programming abstraction adapted to packet parsing, matching, and alteration, *i.e.,* equivalent to the one exposed by the P4 programming language [60].

3. **Precision** Destined to diagnose transient latency issues at small timescales, OP4T must be able to perform timestamping with a precision in the order of the microsecond at worst.

4. **Performance** When replacing a regular server NIC, OP4T must not introduce any performance limitation in terms of achievable throughput or packet rate.

5. **Flexibility** Like most debugging, understanding transient latency spikes in a data-centre can be a complex, and interactive process, *i.e.,* can require changes in the program defining packets to timestamp and alterations to perform. Therefore, those changes must be possible, with minimal disruption in network operations. For example, a full reprogrammation of the FPGA board is not acceptable, as it would require reloading the network interface driver. From a network operation perspective, this is highly disruptive.

The main contributions of this chapter are (i) the design of OP4T as an architecture following those principles, (ii) the implementation of OP4T as an open-source hardware design, usable on the NetFPGA SUME board [75], along with a high-performance network interface driver, (iii) an experimental evaluation of the precision achievable by OP4T for a synthetic traffic pattern and a typical P4 program.

3.1 Related Work and Limitations

In this section, an overview of existing open-source solutions for packet timestamping is given. These solutions are analysed on two particular aspects: performance, and programmability. Other aspects and limitations of the packet timestamping capabilities of state-of-the-art network testers are already detailed in [74].

3.1.1 Performance

To analyse fine-grained latency variations, it is necessary to be able to timestamp, with high-precision, all the packets in a given stream. Moon-

Gen [76] is an open source network tester, exploiting the Precision Time Protocol (PTP) support in commodity Network Interface Cards (NIC), to perform accurate packet timestamping. However, the Application Programming Interface (API), exposed by such a NIC, exposes packet timestamps in an internal register, which is to be read and cleared by the driver each time a timestamp is to be retrieved. This severely limits the achievable packet rate in case all packets must be timestamped.

OSNT is capable of altering the received packets with a timestamp, inserted at a programmable position in the packet. The packet is then transmitted to the host server via Direct Memory Access (DMA), and the timestamps can be retrieved by parsing the received packets. However, preliminary experiments showed that the DMA core used by OSNT has insufficient performance to allow capturing a packet stream of 1.5 Gbps, which is only a fraction of the 10Gbps line-rate supported by the used NetFPGA-SUME board.

FlueNT10G timestamps packets in a way similar to OSNT, but rely on the Xilinx DMA/Bridge Subsystem for PCI Express v3.1 DMA core, which provides sufficient performance to saturate the PCI-Express bus, and therefore, allows capturing all the timestamped packets of a 10 Gbps stream.

3.1.2 Programmability

FlueNT10G offers a level of programmability by providing a software framework allowing network testing automation. However, the programmability of the hardware design itself is limited to specifying a list of MAC addresses, used to filter ingress packets. OSNT has more advanced filtering rules: it allows specifying a list of flows to timestamp, determined by IP-and-port-based packet matching rules.

3.2 Hardware Architecture

Figure 3.1 depicts the architecture of OP4T, independently from the underlying hardware target. In this section, the flow of a packet through this architecture is described, with a focus on timestamp acquisition, and the programmable packet processor.

3.2.1 Packet Flow

Figure 3.1 represents the different blocks traversed by each packet. Along with its data, packet *metadata* is transported between the different blocks.

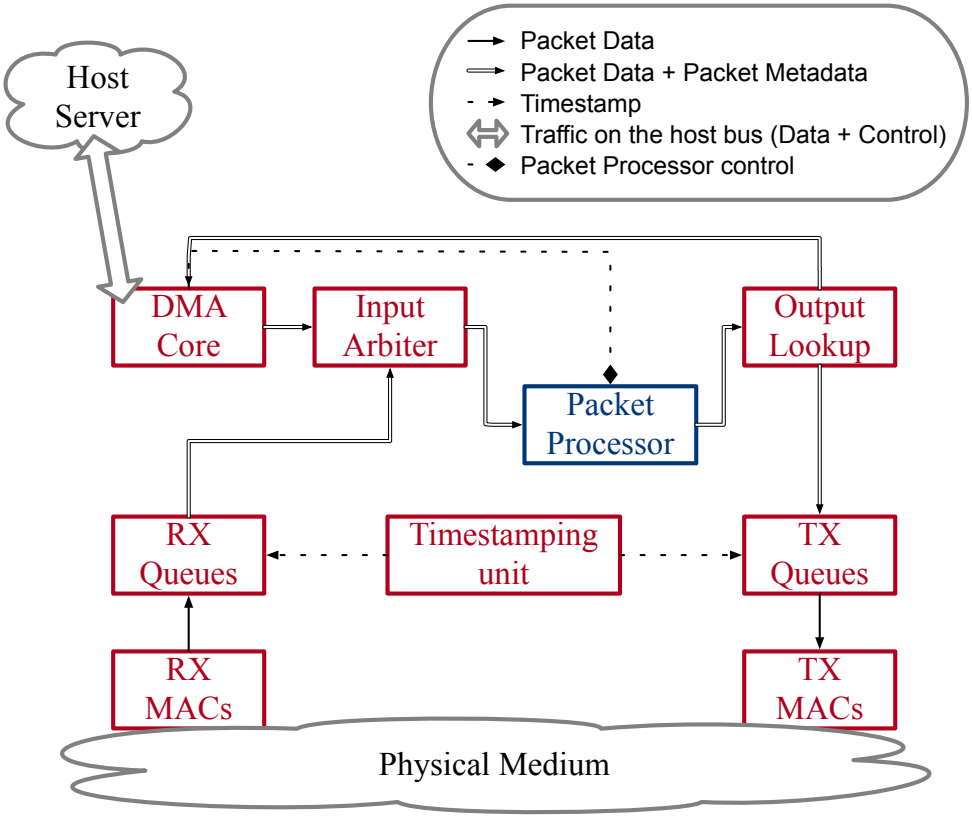


Figure 3.1: Abstract hardware architecture of OP4T. Red: elements from the static design. Blue: user-programmable packet processor.

Metadata includes information related to the packet, and which is to be shared across blocks, *e.g.,* the source and destination physical ports.

As OP4T mimics the behaviour of a network interface, a packet entering the system can originate either from the host server, or from the physical medium. In the first case, packet data and metadata are retrieved from the host memory by the DMA core of figure 3.1. In the second case, the packet is received over one of the physical mediums connected to OP4T, by the corresponding Reception Medium Access Control core (RX MAC). The packet data is then enqueued into a reception queue (RX Queue), along with metadata, generated on-the-fly. In both cases, packet metadata includes the *source* of the packet, *i.e.,* whether it was generated by the host server, or the identity of the reception physical medium.

Regardless of its source, the packet is then transmitted to an input arbiter with multiple packet inputs and one output. The input arbiter selects a packet from one of its inputs, and transmits it to the packet processor. It

alters the packet data and metadata by executing user-specified operations. In particular, a *destination* is added to the metadata. The packet is finally transmitted to the output lookup block, which, depending on the destination, routes it either to the DMA core for transmission to the host server, or to one of the transmission queues (TX Queues), for transmission over one the connected physical mediums.

3.2.2 Timestamp Acquisition

Timestamping is performed in two places in the design: at the ingress of the RX Queues, and at the egress of the TX Queues. Timestamping as closely to the MACs as possible eliminates the jitter introduced by, *e.g.,* the different sources contending at the input arbiter, or by the packet processor.

At the RX Queue, a timestamp is generated for each incoming packet, and inserted in the packet metadata for later consumption by the packet processor. However, that is not applicable at the TX Queue, as metadata are lost upon packet transmission over the physical medium. Therefore, a timestamp acquired at the TX Queue must be inserted in the packet data. To avoid altering all the packets flowing through the design, the packet metadata includes a flag, set by the packet processor, and indicating whether a timestamp should be added by the TX Queue. Moreover, if that flag is set, the packet metadata must also include the data offset where the timestamp should be inserted.

3.2.3 Reconfigurable Packet Processor

The packet processor contains user-defined logic, performing data and metadata alteration. That logic is expected to set the destination information in the metadata, so that the output lookup block can route the packet. Typically, to mimic the behaviour of a NIC, the user-defined logic should read the source from the metadata, and route packets originating from a physical medium towards the host, and those originating from the host to the corresponding physical medium.

Runtime flexibility is brought to the packet processor by *control-plane*, and by *partial logic reconfigurability*. As represented on figure 3.1, control-plane enables the host server to push stateful information into the packet processor at runtime, *e.g.,* traffic matching rules, specifying packets to be altered with timestamps. Partial logic reconfigurability is a feature of some FPGAs, enabling updates of part of the programmed logic, without erasing and resetting the whole design. In OP4T, the packet processor is partially

reconfigurable, enabling live updates, with minimal disruption from the perspective of the host. When performing live latency debugging, this allows a network operator to push a packet processor logic, specifically tailored to the current debugging scenario, without a full reset of what appears as a network interface to the host server.

In the remainder of this chapter, the term *static design* designates all the components of figure 3.1 that are not reconfigurable at runtime, *i.e.,* all the elements at the exclusion of the packet processor.

3.3 Implementation

The hardware architecture described in section 3.2 was implemented on the NetFPGA SUME FPGA board [75][1]. As much as possible, this implementation also relies on open-source Intellectual Property (IP) cores, as detailed in the following.

3.3.1 Overview

The presented implementation of OP4T is derived from OSNT-SUME, *i.e.,* the adaptation of OSNT to the NetFPGA-SUME board. Specifically, the RX and TX MACs, timestamping unit, input arbiter and output lookup block from figure 3.1 are reused from OSNT-SUME. The RX Queues and TX Queues blocks are derived from those used in OSNT-SUME, and were modified to implement timestamping as described in section 3.2.2. Following the OSNT-SUME implementation, the block interconnections from figure 3.1 are implemented by AXI4-Stream buses for packet data and metadata transport, an AXI4-Lite bus for the control-plane, and the PCI Express bus for the interface with the host server.

Because of its performance limitations, the DMA core included in OSNT-SUME and NetFPGA-SUME was replaced with a custom implementation, which consists of a wrapper around the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) IP core, provided as part of the Xilinx toolchain.

Finally, the P4-NetFPGA workflow and source code [77] are partially reused and adapted to allow the creation of custom packet processors in P4.

[1] In 2020, the NetFPGA SUME board is the de facto standard prototyping platform in the network research community.

3.3.2 DMA Core integration

The proposed implementation of OP4T uses the XDMA IP core to provide high-performance packet transmission and reception over PCI-Express (PCIe). Specifically, XDMA provides Card To Host (C2H) and Host To Card (H2C) *channels*. Those appear to the host as queues of read (for a C2H channel) or write (for a H2C channel) *DMA operation descriptors*. Each descriptor is first enqueued by the software, then dequeued by the XDMA hardware, which finally executes the associated read/write DMA operation. On the hardware design, channels appear as AXI4-Stream (for the data-plane) or AXI4-Lite (for the control plane) input (for a C2H channel) or output (for a H2C channel) ports belonging to the XDMA core.

Because the host is to be exposed the semantic of a network interface, each received packet is mapped to a read DMA operation, and each transmitted packet, to a write DMA operation. That is performed in the proposed OP4T implementation by a Data Plane Development Kit (DPDK) Poll-Mode Driver, communicating with XDMA over PCIe, and translating DMA operations into packets.

3.3.3 P4 Packet Processor and Partial Reconfiguration

To facilitate the implementation of custom user-logic for the packet processor, the proposed design extends the Xilinx Vivado Partial Reconfiguration flow [78], as depicted in figure 3.2. First, the static design is synthetised **(a)**, into a netlist. Then, a first flavor of the user-logic, denoted by **packet processor 0** and written in P4, is translated into Register-Transfer Level (RTL) code **(b)** by the Xilinx P4-SDNet and Xilinx SDNet compilers[2], following the P4-NetFPGA workflow [77]. The generated RTL code is then synthetised **(c)** into a netlist, which is combined with the netlist of the static design. The resulting netlist is placed and routed **(d)**, finally forming an implementation of OP4T with Packet Processor 0 (OP4T-PP0).

To enable partial reconfiguration of the packet processor, *i.e.,* replacement of packet processor 0 with another P4-based user logic, the implementation of OP4T-PP0 is first stripped from all placement and routing information related to packet processor 0 **(e)**, resulting in the implemented static design. Then, another P4 code, denoted by **packet processor 1**, is translated into RTL code $\mathbf{(b_1)}$, synthetised $\mathbf{(c_1)}$, and combined with the implemented static

[2]This toolchain is described by documentation available at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1252-p4-sdnet.pdf` and `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf`

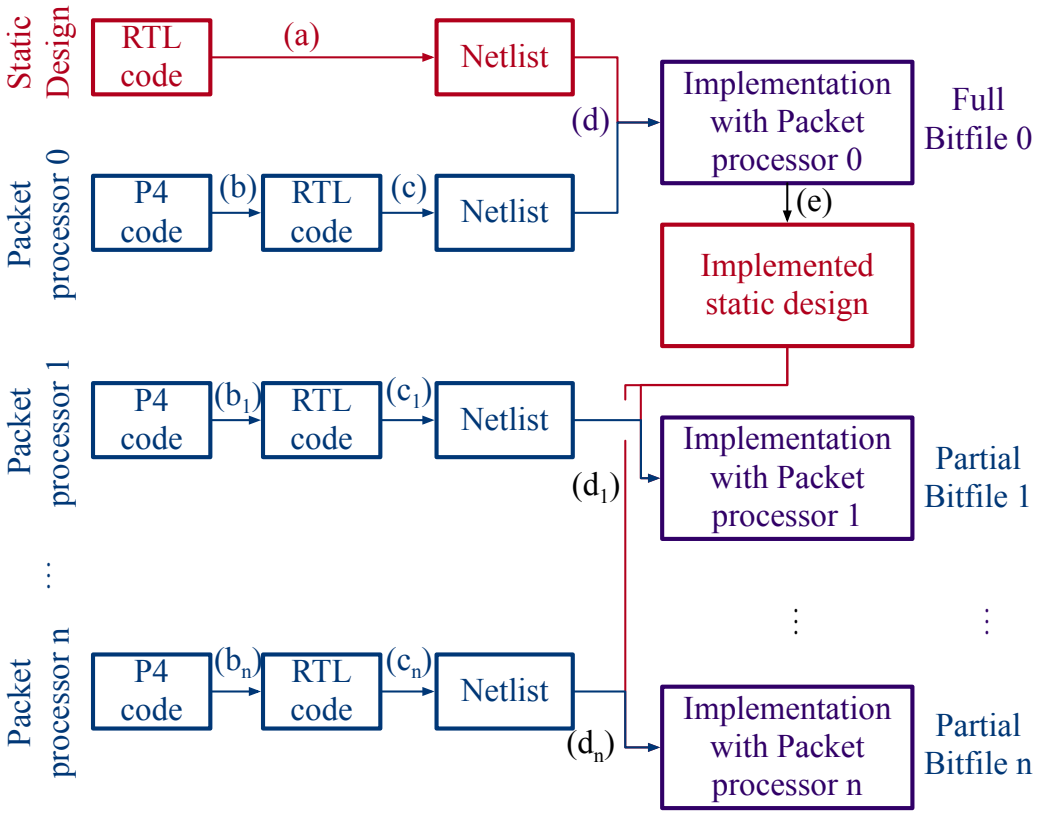


Figure 3.2: Vivado Partial Reconfiguration flow applied to implementing P4 packet processors for OP4T. Red: elements of the static design. Blue: elements of the packet processor. Purple: output products combining elements from the static design and from the packet processor.

design $(\mathbf{d_1})$ for placement and routing. The result of that operation is an implementation of OP4T with Packet Processor 1 (OP4T-PP1), fully compatible with that of OP4T-PP0, *i.e.,* placement and routing of the elements of the static design are identical in both implementations. Therefore, partial reconfiguration is possible by reprogramming, at runtime, only the FPGA area corresponding to the packet processor. This operation can be repeated for as many additional P4-based user logics as needed, as shown in figure 3.2.

Placement and routing of OP4T with additional packet processors are constrained by the placement and routing of OP4T-PP0 **(d)**, which are constrained by the complexity of packet processor 0 itself. Therefore, when using the workflow of figure 3.2 for implementing multiple configurations OP4T-PP0, OP4T-PP1, ..., OP4T-PPn, packet processor 0 should be chosen so that OP4T-PP0 is the most challenging configuration for the placement and routing engines.

3.3.4 Discussion

The workflow detailed in section 3.3.3 enables implementing multiple versions of OP4T, each with a a different packet processor. Moreover, partial reconfiguration enables switching from one version to the other, without disrupting the static design, thus, requiring neither resetting the DMA core, nor impacting the network interface exposed to the host server. However, two main difficulties arise when implementing partial reconfiguration.

Firstly, it requires *floorplanning*, *i.e.,* the FPGA physical resources allocated to the static design and to the packet processor must be determined prior to placement and routing. Although automated tools can assist floorplanning [79–81], it still requires prior knowledge of the resource utilisation induced by the packet processor. Therefore, once floorplanning is performed, the thereby provisioned resources limit the complexity of any future packet processor.

Secondly, choosing packet processor 0 so that OP4T-PP0 is the most challenging design to place and route implies prior knowledge of all future packet processors, which is not necessarily possible in the scenarios targeted by OP4T, *e.g.,* generating an ad hoc packet processor to diagnose a given latency issue, in a production data-centre, without fully reprogramming the FPGA. The proposed implementation avoids that issue by explicitly registering all the inputs and outputs between the static design and the packet processor, at the cost of increased latency.

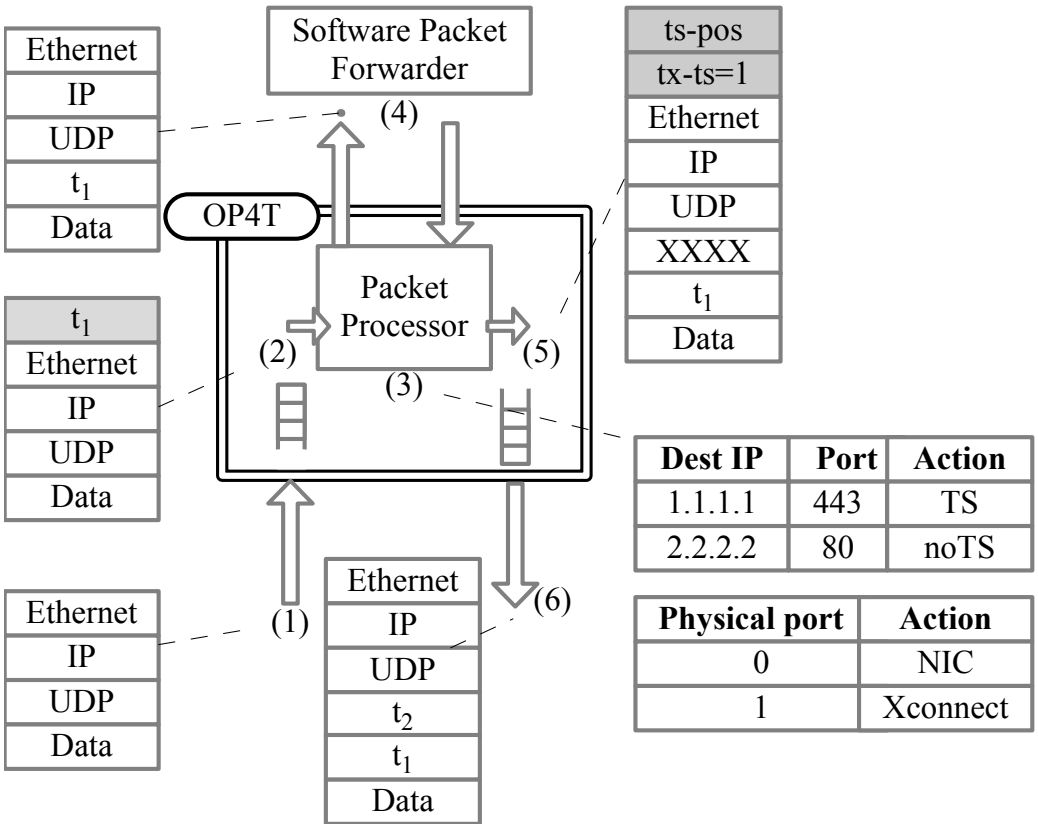


Figure 3.3: Packet Flow when testing a software switch. Greyed information is metadata transported with a packet. t_1 is the timestamp sample upon packet reception from the network. tx-ts is the flag indicating that the TX Queue must add a timestamp upon transmission, at the position indicated by ts-pos.

3.4 Case Study: OP4T for Software Switch Testing

In this section, the OP4T implementation from section 3.3 is used in a latency evaluation scenario, different from network testing: evaluating the latency introduced by a software packet forwarder itself, *i.e.,* only the latency introduced by packet processing and PCIe-based DMA. To that end, a packet processor is specified, inserted in the OP4T design, eventually forming the OP4T for Software Switch Testing (OP4T-SST) configuration.

3.4.1 Scenario

When evaluating the latency of a Packet Forwarder (PF), a network tester generates a packet stream (testing stream), transmits it to the PF and monitors the packets forwarded by the latter. The Round Trip Time (RTT) obtained by comparing transmission and reception timestamps at the network tester provides an evaluation of the latency of the PF. While that measurement is impacted by the packet delay variations due to the network path, those can be considered as negligible in a controlled infrastructure (part of a network testing setup). Consequently, the measurements obtained are precise.

However, in a real deployment hosted in a data centre running production applications, testing traffic is subject to non-negligible packet delay variation, due to, *e.g.,* congestion and queueing occurring in the data-centre switching fabric. To accurately evaluate the latency of a software packet forwarder — typically, a Virtual Network Function — deployed in such a data-centre, it is necessary to timestamp packets upon reception and transmission, at the network interface. If the latter is based on OP4T, such timestamping can be implemented by a custom packet processor. In this case study, the testing stream to be selected by OP4T for timestamping, is a User Datagram Protocol (UDP) stream.

3.4.2 OP4T-SST Packet Processor

The behaviour of the packet processor used in OP4T-SST is represented in figure 3.3. When a packet is received (1), OP4T generates a timestamp t_1, inserted in the packet metadata (2). The packet processor then parses the packet's headers, and performs a table lookup (3). The destination Internet Protocol (IP) address is matched against a table (created by the host server, through the control plane), to determine whether the packet belongs to a stream of interest to be timestamped. This lookup can either yield a TimeStamp (TS) or a do not TimeStamp (noTS) action. If the TS action is matched, then, t_1 is inserted between the UDP header and the data. Otherwise the packet is not altered. Then, the processed packet is sent to the host through the DMA core (4). As a Software Packet Forwarder is running on the host, the packet is sent back to OP4T, and, the packet processor performs the same table lookup as before. If the TS action is matched, the packet is altered to provision zero-ed bytes immediately after the UDP header, and the packet metadata is altered, so that, upon transmission, the TX Queue block replaces the provisioned bytes with a transmission timestamp t_2 (5). Finally, the resulting packet contains, just after the UDP header, two timestamps

t_2 and t_1, whose difference evaluates the latency introduced by the Software Packet Forwarder.

3.4.3 Precision and Cross-connect

As a consequence, that measured latency includes the delay introduced by the packet processor itself. Due to internal pipelining, this delay may not be constant, impacting measurement precision.

To evaluate the loss thereof, the OP4T-SST packet processor has a cross-connect (Xconnect) feature. When a packet is received from the network, it is not necessarily sent to the DMA core. Instead, a lookup is performed on another table, different from the one mentioned in section 3.4.2 , as shown in figure 3.3. If the Xconnect action is matched in this table, the packet is directly transmitted back to the network, without going through the Software Packet Forwarder. Moreover, in that case, if the packet is determined to match the TS action, t_2 and t_1 are both appended after the UDP header, thus, providing an evaluation of the base latency introduced by the packet processor.

3.5 Evaluation

This section presents the process and results of a quantitative evaluation of OP4T-SST.

3.5.1 FPGA Resource utilisation

As stated in section 3.3.4, the implementation of OP4T requires explicit floorplanning. The conducted experimental evaluation used the floorplan depicted in figure 3.4. While the packet processor spans over almost all the available slices, actual FPGA resource usage is summarised in table 3.1. Overall, OP4T-SST occupies approximately one third of all FPGA resources. Specifically, the packet processor only uses half the resources it is allocated by floorplanning, making the used floorplan suitable for designing additional, more complex packet processors.

3.5.2 Experimental Setup

The study aims at evaluating the precision of OP4T-SST, as well as exploring how features of the testing traffic impact the latency necessarily introduced by the design.

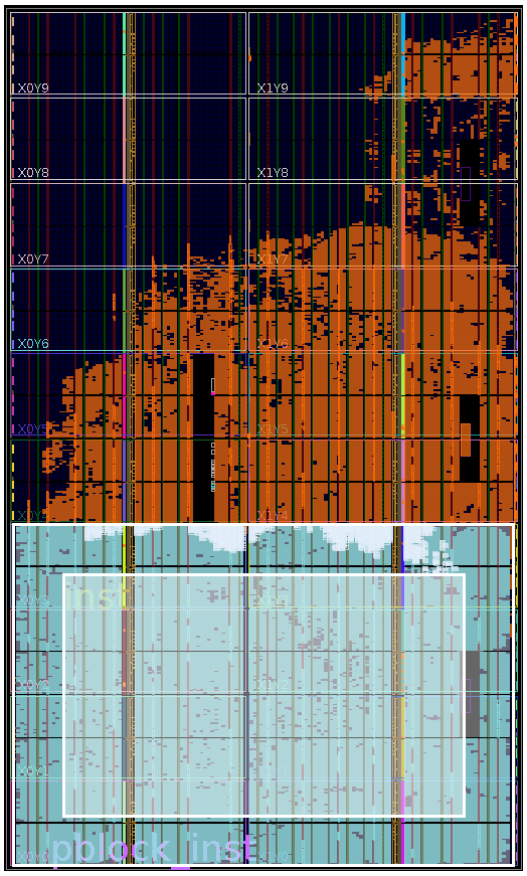


Figure 3.4: Floorplanning for OP4T with the packet processor defined in 3.4. The lower part is reconfigurable and allocated to the packet processor (blue), which almost fills it. The upper part is allocated to the static design (orange). In white are represented inteconnection points between the reconfigurable packet processor and the static design.

Table 3.1: Resource Utilisation

Resource	Static Design		Packet Processor		Total	
	Used	Usage	Used	Usage	Used	Usage
Slice LUTs	64130	24.44%	86643	50.73%	150773	34.80%
Slice Registers	84186	16.04%	152297	44.58%	236483	27.29%
Block RAM	266.5	30.28%	268.5	44.51%	535	36.39%

The software packet forwarder under test is the Vector Packet Processor (VPP) [82]. To obtain precise and reproducible results, the purpose in this evaluation is the latency experienced by constant-rate packet streams with a stable period. Contrary to state-of-the-art network testers, OP4T does not integrate any packet generator, and the method developed in MoonGen is used for generating a testing stream with a stable period. That method consists of transmitting, at line-rate, alternately one packet from the testing stream, and a certain number of other packets, crafted so as to be dropped by a packet switch before reaching the receiver (*e.g.,* with a bad Cyclic Redundancy Check(CRC)). That last number of packets determines the period achieved by the generator.

The experimental setup consists of a packet generator, a packet monitor, and a server running VPP and hosting a network interface implemented as a NetFPGA-SUME board programmed with OP4T-SST. A testing stream is transmitted by the generator, traverses the server through OP4T (as depicted in figure 3.3), and is finally received by the packet monitor. The latter then estimates the latency incurred by each packet by computing the difference between the two timestamps t_1 and t_2.

Finally, OP4T-SST and/or VPP must be configured to forward the stream received from the packet generator to the packet monitor. Four ways were experimentally implemented to achieve that goal: (i) using the Xconnect feature of OP4T-SST, effectively bypassing VPP and yielding a baseline latency estimation, (ii) using the layer-2 patch feature of VPP (iii) using the layer-2 cross-connect feature of VPP (iv) using VPP as a layer-3 packet switch with an appropriately configured routing table. As those three last VPP-based configurations rely on code paths of increasing complexity, comparing the measured latencies in those configuration evaluates the ability of OP4T-SST to detect fine-grained software behaviour.

3.5.3 Results

Figure 3.5 summarises the different experimental results. The baseline experiment shows that, the latency incurred by traversing OP4T has a negligible standard deviation (around 4 nanosecond). This number is difficult to improve, as the used implementation is clocked at 250MHz, *i.e.,* with a 4 nanosecond period. Moreover, the measured baseline latency only depends on the packet sizes, not on the packet rate. Dependency on packet sizes is explained by store-and-forward packet transmissions at the ingress and egress of the hardware design, which necessarily add an extra-latency proportional to the packet size.

The results for the three VPP-based configurations realistically illustrate

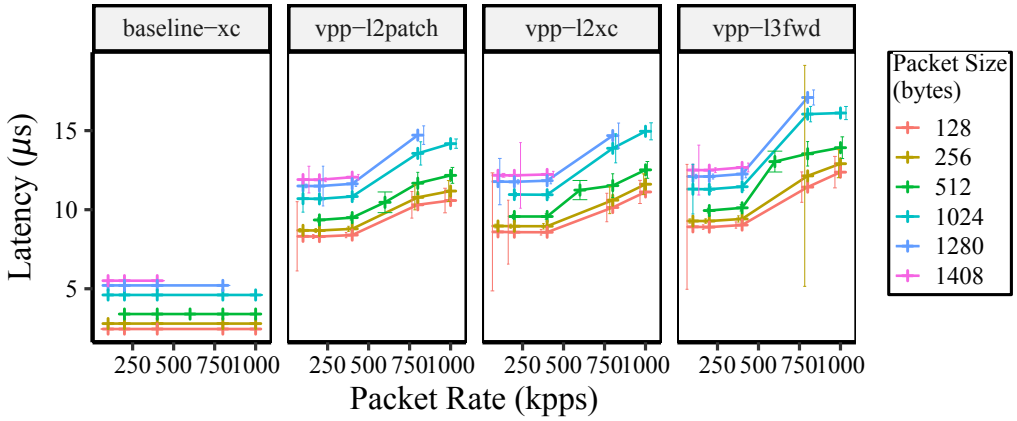


Figure 3.5: Measured average latency experienced by testing streams of various packet rates and sizes. The server traversed by the stream is configured (from left to right): (i) with the OP4T-SST Xconnect feature, (ii) with the layer-2 patch VPP feature, (iii) with the layer-2 cross-connect VPP feature, (iv) with VPP configured as a layer 3 packet switch. Error bars are centred around the average and their height is twice the standard deviation of the latency.

the internal behaviour of VPP. First, the latency increases with the complexity of the involved code, with a clear difference between layer-3 forwarding and layer-2 cross-connect. Second, the standard deviation also increases with the complexity of the code, which is explained by an increased number of sources of packet processing time variations (cache-misses, software or hardware preemption). Finally, in a given configuration and for a given packet size, latency is constant when increasing the packet rate, until a clear threshold (400000 packets per second), beyond which latency is linear. This is an experimental confirmation of batched packet processing occurring in VPP.

3.6 Conclusion

OP4T specifies an open programmable architecture, capable of high-precision packet timestamping, in situ, *i.e.,* deployed in a data-centre. To achieve that goal, OP4T is designed to be usable simultaneously as a network interface, transmitting and receiving production traffic, and as a partially reprogrammable packet timestamp acquisition device, altering selected packets by the adjunction of reception and/or transmission timestamps. The

programmability of OP4T is key to debugging complex latency issues, as it brings the ability to interactively refine the packet timestamping logic, without disrupting the exposed network interface.

This architecture was implemented on the NetFPGA-SUME board, relying on open source IP cores derived from the NetFPGA-SUME, P4-NetFPGA, and OSNT-SUME projects. Specifically, programmability was achieved by the joint use of the P4 programming language, and partial logic reconfigurability provided by modern FPGAs. The obtained open-source implementation was shown to achieve timestamping with a precision in the order of a single clock cycle, and was shown to be precise enough to measure fine-grained properties of a software packet forwarder such as VPP, which is a synthetic example of data-centre application.

As mentioned in chapter 1, packet transport and processing in the context of media production are specific cases of latency sensitive applications. In particular, the work presented in this chapter can be reused to qualify the regularity of SMPTE 2022-6 or SMPTE 2110 flows. The next chapter proposes a deep dive into the *definition* of such regularity, proposes methods to ensure it (through *packet-pacing*), and leverages part of the work developed in this chapter to experimentally evaluate them.

Chapter 4

High-Accuracy Packet Pacing on Commodity Servers for Constant-Rate Flows

As described in chapter 1, because the packet-based SMPTE 2022-6 and SMPTE 2110 media production standards are derived from unidirectional SDI-based media transport, they formally (for SMPTE 2110) or informally (for SMPTE 2022-6) mandate that packets are delivered to a receiver with a certain regularity, which is formalised in this chapter. Furthermore, it develops methods to achieve *packet pacing* on commodity servers, and, therefore, demonstrates that those are able to transmit packets with a regularity compatible with the aforementioned standards, and hence, are usable to implement a media production setup relying on commodity hardware.

Background

In packet-switched networks, the packet transmission rate needs to be controlled so that the receive buffer never overflows. Known as *flow control*, that task is usually achieved asynchronously, *e.g.,* with the stop-and-wait Automatic Repeat reQuest (ARQ) protocol (introduced in [83]), or with a sliding-window-based algorithm, *e.g.,* as in the Transmission Control Protocol (TCP) [84]. For either of those, the receiver sends an explicit signal, authorising the sender to transmit more data. TCP achieves flow control by permitting a maximum number — the current *receiving window* — of transmitted and unacknowledged data. A TCP Acknowledgment (ACK) decreases the amount of unacknowledged bytes and thus acts as authorisation for the sender to transmit subsequent data.

Some applications will consume packets from their receive buffer at a

given rate, independently from the state of the buffer. An example hereof is a video player, which expects to always have enough packets available to build the next video frame. If incoming packets are late, these applications might try to consume a packet from an empty buffer, a condition known as *buffer starvation* – which, in the video player example, results in a visually unsatisfactory user experience.

The design of the system, the sender, and the receiver, should prevent this from happening, *i.e.,* each packet in such a stream must be received *before* a certain time. This is different from flow control, which seeks to delay transmission by the *sender* until *after* it has been confirmed that there is buffer space available by the receiver.

Towards Software Pacers

If a sender transmits, and receiver consumes, packets at the exact same rate, the receive buffer will be subject to neither overflow nor to starvation – assuming that the delay introduced by the network between sender and receiver is *close* to constant. As mentioned in chapter 1, that is the approach adopted in standards defining packet-based transport for media production such as SMPTE 2022-6 and 2110, as those assume Constant-Rate (CR) packet transmission.

Therefore, as those standards are intended as enablers for the broadcasting industry to migrate media processing from dedicated to commodity hardware, the latter must be able to generate and carry sufficiently regular CR streams.

Furthermore, and as mentioned in chapter 1, migration from *"all media-dedicated hardware"* to *"all software running on commodity hardware"* includes an intermediate state with SDI-to-IP and IP-to-SDI *gateway devices*. As a transition technology, gateway devices may have (for reasons of cost-containment) limited memory, *i.e.,* limited receive buffers. Consequently, upon reception of an *insufficiently regular* stream, such buffers are potentially exposed to overflow and starvation.

This motivates the design of a *packet pacer* – a system, which buffers an incoming, and potentially *insufficiently regular*, packet stream and releases the packets *sufficiently regularly* for a CR receiver to never be subject to buffer overflow, nor to starvation. In support of transitioning from dedicated, to commodity, hardware, such a *packet pacer* should be a *generic* software solution, requiring also only *generic* hardware.

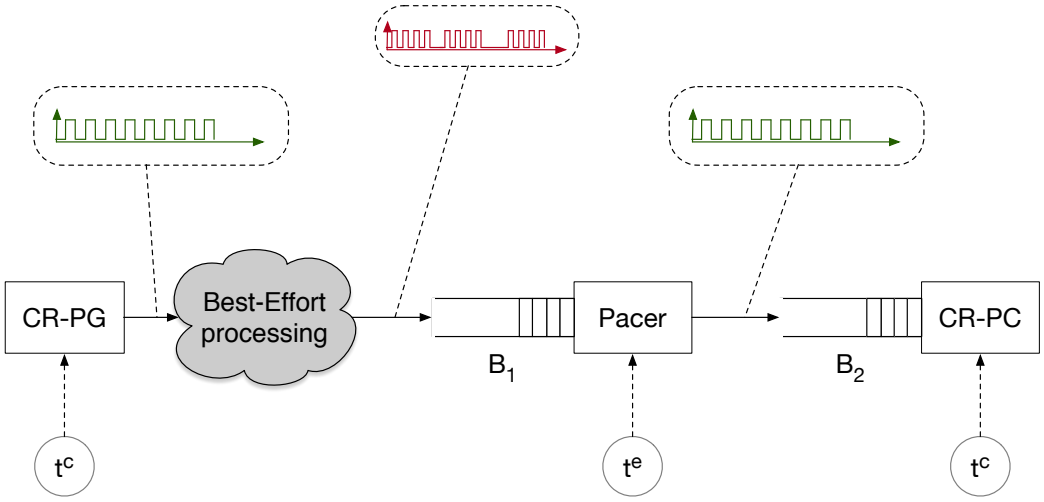


Figure 4.1: System overview: A stream of packets, generated by a Constant-Rate Packet-Generator (CR-PG), undergoes some Best-Effort processing, and hence a loss of regularity. Those packets are buffered into B_1 by the Pacer, which transmits them, with sufficient regularity, so that the receive buffer B_2 of a Constant-Rate Packet-Consumer (CR-PC) never undergoes overflow or starvation. Packet transmissions (by the CR-PG) and consumptions (by the CR-PC) occur at times, sorted into a sequence t^c.

Related Work

Previous work related to packet pacing can be categorised into techniques to avoid congestion control and reduce queueing delays, into specifically scheduling of packet transmission for video-streaming, and into implementability considerations for real-time packet transmission.

Techniques to avoid congestion and reduce queuing delays

In [85], the effects of pacing on TCP flows are shown to bring higher fairness at the cost of lower overall throughput, especially as the number of concurrent flows increases. Further, [86] identifies a *point of inflexion* in the number of flows, above which pacing reduces throughput. This is explained by a synchronisation phenomenon when TCP flows experience simultaneous congestion window reductions, leading to inefficient network usage, first described in [87]. Pacing is also a building block of Bottleneck Bandwidth and Round-trip time (BBR) [88], a congestion control algorithm for TCP that reacts to congestion by reducing the pacing rate, instead of by solely

reducing the TCP congestion window. In [89] and [90], dynamic pacing schemes are proposed to improve transport-layer performance in small-buffer networks and reduce congestion. Queue-Length Based Pacing (QLBP), described in [89], uses the number of buffered packets to determine the pacing rate. In [90], a poly-logarithmic-complexity online pacing algorithm is shown to reduce short-timescale burstiness, while still transmitting each packet before a given deadline.

Video streaming

In [91], the problem of scheduling packet transmissions so that a CR receiver experience neither buffer overflow nor starvation is formulated in the framework of *network calculus*, and the set of feasible packet departure curves is determined, given the characteristics of the receiver and the network. In [92], this problem is addressed for stored video, and an optimal packet transmission schedule is constructed, given a non-necessarily-constant packet consumption profile at the receiver. This is applicable, *e.g.,* to Variable Bit-Rate (VBR) video streaming.

Implementability

Given the real-time nature of the pacing task, implementing it as a piece of software proves challenging. In [93], the scalability of traditional implementations of pacing (e.g. the Linux Hierarchical Token Bucket (HTB) or the Fair Queue (FQ) queueing discipline (qdisc)) is shown to be limited for large number of flows to be paced, due to the intrinsic per-flow cost.

Software pacing without relying on timers is proposed in [94], by introducing a concept of *gap-frames*: interleaving carefully-sized IEEE 802.3 flow control frames (as defined in [43]) and packets from a flow of interest. IEEE 802.3 flow control frames are dropped by a packet switch on the network path, resulting in pacing of the flow of interest. In [95], a timer-based implementation is introduced, compared to the gap-frame approach and is experimentally shown to outperform the latter in terms of accuracy. That conclusion is challenged in [76], which uses the gap-frame approach to implement rate-control in a packet generator and evaluates the resulting pacing accuracy to be higher than in [94] and [95]. A possible explanation is that the implementation in [76] relies on the Data Plane Development Kit (DPDK) [96], which is designed to provide higher and more predictable performance than is the Linux network stack, which was used in [94] and [95].

Statement of Purpose

The objective of this chapter is to propose a software-based packet pacer, able to run on commodity hardware, able to accept an *insufficiently regular* packet stream, and able to return a *sufficiently regular* packet stream, suitable for any CR receiver – including receivers expecting an SMPTE compliant packet stream. This is illustrated in figure 4.1.

Further, given that the aforementioned production *gateway devices* are opaque packets consumers, this chapter – unlike *e.g.,* [91] – assumes that the pacer sees the Constant-Rate Packet Consumer (CR-PC) as an opaque component, *i.e.,* with an unknown packet-consumption curve.

The required real-time properties make running a pacer as software on commodity hardware challenging. This chapter analyses and formalises those challenges. This chapter also generalises the previously discussed use of gap-frames [76, 94] for accurate packet scheduling, into the notion of a Pacing-Assistant – which enables abstract expression and analysis of packet pacing algorithms.

Finally, within this framework, software pacing algorithms are analysed, are implemented, and are subjected to exhaustive experimental tests, confirming the viability of the postulated approach.

Chapter Outline

The remainder of this chapter is organised as follows. Section 4.1 formalises the pacing problem and section 4.2 discusses the limitations of a pure-software approach. Section 4.3 introduces Pacing-Assistants (PA), as a form of hardware-assistance, and proposes algorithms using it. Those are supported by a theoretical analysis provided in section 4.4, and are shown to be effectively implementable on general-purpose hardware in section 4.5. Section 4.6 presents the associated experimental evaluation. Section 4.7 discusses the obtained results and their practical impact for media production. Section 4.8 concludes this chapter.

4.1 System Model

While the *theoretical* objective of packet pacing is to generate a *perfectly* constant-rate packet stream, the formalism of *time sequences* is introduced in this section, to provide a *practical*, quantitative definition of a *sufficiently-regular* constant-rate packet stream.

4.1.1 Time sequences

The regularity of packet stream is only described by the sequence of transmission times. Defined hereafter, the formalism of *time sequences* enables describing any sequence of recurring events.

Definition. *A **time sequence** is a nondecreasing sequence of times* $t = (t_i)_{i\in\mathbb{N}}$*, such that* $\lim_{n\to+\infty} t_n = +\infty$*. The abstract event happening at time* t_i *is called the* i*-th **cycle** of* t*, or the* i*-th* t*-cycle.*

Definition. *The following metrics are used to quantify the potential periodicity of a time sequence.*

- *The **period** of* t *is, when it exists,* $T := \lim_{n\to+\infty} \frac{t_n}{n}$*. This definition is equivalent to the average duration between two consecutive events,* i.e., $T := \lim_{n\to+\infty} \frac{1}{n}\sum_{i=1}^{n} t_i - t_{i-1}$.
- *The **frequency** of* t *is the inverse of the period, when it exists,* $f := \frac{1}{T}$.
- *The **Peak Period jitter** of* t *is:*

$$J_p(t) = \limsup_{i\geq 0} |T - (t_{i+1} - t_i)|$$

- *The **Asymptotic Long Term (ALT)** jitter of* t *is:*

$$J_{ALT}(t) = \inf_{\substack{i_0\in\mathbb{N}\\ u\geq 0}} \sup_{\substack{i\geq i_0\\ t_i\geq u}} |T(i-i_0) - (t_i - u)|$$

$J_P(t)$ asymptotically measures how far the duration between two consecutive cycles, $t_{i+1} - t_i$, is from the period T. The definition of $J_{ALT}(t)$ is interpreted by comparing t with a perfectly-regular time sequence t' whose i-th cycles occur at $t'_i = T \times i$. t and t' have the same period T. As it is perfectly-regular, t' can be used to measure the current time with a granularity of T, provided that a well-known number i of t'-cycles have elapsed. $J_{ALT}(t)$ asymptotically quantifies the maximal error on that measurement if t is used in place of t'. Therefore, $J_{ALT}(t)$ provides a certain measure of the irregularity of t when compared with the perfect time sequence t' of same period T.

In the rest of this chapter, t^p is the time sequence corresponding to the packet transmission times by the pacer of figure 4.1. t^c is the time sequence corresponding to the packet production times by the Constant-Rate Packet

Generator (CR-PG), and consumption times by the CR-PC[1]. The periods and frequencies are respectively denoted by T^p, T^c, f^p and f^c.

4.1.2 (b, f)-paced streams

In the following, a definition of a *sufficiently regular* constant-rate packet stream is given as a property, relating the time sequence t^p associated with packet transmission, with a buffer size b and a packet consumption rate f, both associated with the receiver. It is assumed that there is no delay between the sender and a receiver. Upon reception of the i-th packet, an infinite-sized buffer at the receiver would contain $i - ft_i^p$ packets. That motivates the following definition:

Definition. *A system sends a (b, f)-paced stream if the packet transmission time sequence $(t_i^p)_{i\in\mathbb{N}}$ are such that there exists a time, t_0, and an integer, i_0, satisfying:*

$$\forall i \geq i_0 : t_i^p \geq t_0 \implies 0 \leq (i - i_0) - f \times (t_i^p - t_0) \leq b \tag{4.1}$$

i.e., a virtual receiver, consuming packets at a rate f and with a b-sized input buffer, may choose an integer number i_0 and a time t_0, so that, dropping the i_0 first packets and only starting consumption after time t_0 prevents any overflow or starvation.

This definition reflects only the asymptotic behaviour of packet transmissions, *i.e.,* is sensitive neither to the initial variability of the stream, nor to the initial behaviour of the consumer. Moreover, the provided definition is still valid if the delay between the transmitter and the receiver is constant — and not null as initially assumed — because any constant delay can be captured in t_0, *i.e.,* the time packets start being consumed.

Using equation (4.1) for i and $i + 1$, taking the difference, and dividing by f, yields:

Property 1. *If a stream of packet transmission time sequence t^p is (b, f)-paced, then there exists an integer i_0 such that for all integers $i \geq i_0$:*

$$\left| \frac{1}{f} - \left(t_{i+1}^p - t_i^p\right) \right| \leq \frac{b}{f}$$

[1]The assumption here is, that the CR-PC would be able to – was it connected directly to CR-PG, such as would be the case for a video camera and a viewscreen – consume packets at the precise rate at which they were generated. Pacing is intended for compensating for the *best effort* processing variation that occurs in case when CR-PC and CR-PG are *not* directly connected, as in figure 4.1.

i.e., the buffer of a virtual receiver, as in the definition of a (b, f)-paced stream, never experiences variations larger than b between two arrivals.

Property 1 gives necessary conditions — *real-time* constraints — that must be verified if a stream is to be qualified as *sufficiently-regular*. Specifically, the $(i+1)$-st packet can be output neither before time $t_i^p + \frac{1-b}{f}$ nor after time $t_i^p + \frac{1+b}{f}$. Consequently, if a (b, f)-paced stream is to be generated by software, the latter must be executed on a system providing (i) some notion of time consistent with the frequency f, so that software can wait at least $\frac{1-b}{f}$ between two transmissions, and (ii) execution-time guarantees so that a transmission can never occur more than $\frac{1+b}{f}$ after the previous transmission. Those constraints shall be used in section 4.2 to demonstrate the difficulty of pure-software pacing.

For the time sequence t^p associated with the transmission of a (b, f)-paced stream, property 1 only provides necessary conditions, not sufficient ones. As a counter-example, a packet stream transmitted at times $t'_i = Ti + \ln(i)$ verifies property 1 without being (b, f)-paced.

The following property uses the peak period jitter and ALT jitter to provide metrics, quantifying to what extent a stream is constant-rate. In particular, it provides a necessary and a sufficient condition to qualify a (b, f)-paced stream.

Property 2. *t is a time sequence of frequency f, associated with the packet transmission times of a stream.*

1. *If the stream is (b, f)-paced, $J_{ALT}(t) \leq \frac{b}{2f}$.*
2. *Conversely, if $J_{ALT}(t) < \frac{b}{2f}$, then the stream is (b, f)-paced.*
3. *If t satisfies*

$$\exists i_0 : \forall i \geq i_0 : \left| \frac{1}{f} - (t_{i+1} - t_i) \right| \leq \frac{b}{f} \tag{4.2}$$

   then $J_p(t) \leq \frac{b}{f}$.

4. *Conversely, if $J_p(t) < \frac{b}{f}$, then t satisfies equation (4.2).*

Proof. (See Appendix) □

While, as per property 1, a too high value of J_p is incompatible with sufficient regularity, the latter is shown to be equivalent to a low value of J_{ALT} by property 2. J_{ALT} shall therefore be used to verify the correctness of the algorithms developed in this chapter.

4.2 Limitations of a pure software approach

In this section, a software execution model is given, and shown to be rich enough to describe commonly-used commodity hardware. As part of that model, *timers* are defined as components, enabling software to have access to time – which is necessary to transmit a sufficiently-regular CR stream. By inspecting the available timers on a commodity server and the achievable latency, software-pacing is finally proven to be challenging.

4.2.1 Software Execution Model

In this chapter, software execution on a commodity server is modelled *asynchronously*. A server is modelled as a set of programmable components, each considered as a *reactive system*, maintaining an *internal state*, and receiving *notifications* from other components. Software run by a component consists of the specification, for each received notification, of a sequence of actions to be executed by the component. This sequence, called the notification *handler*, consists of actions which are either updates to the internal state, or transmissions of notifications to other components[2].

Not only can such an execution model describe *interrupt-driven* components — as a notification models an interrupt, and a handler models an Interrupt Service Routine (ISR) — but it can also describe components *busy-waiting* for some condition involving an *external state*, and executing some action as soon as that condition is verified. In that latter case, the notification is the external state update, and the handler is the action executed whenever the condition is satisfied. The *latency* of a component is the — generally unknown — execution duration of the handler associated with a received notification.

That execution model describes any commodity server based on an interruptible Central Processing Unit (CPU), and running a preemptive multitask operating system. Considering, *e.g.,* an x86_64 server running a Linux kernel, three types of components can be described, each exhibiting an asynchronous behaviour.

A user-space application

is composed of a sequence of instructions, either updating an internal state (such as registers or private memory), reading from or writing to an

[2]A more formal treatment of that topic is provided by *input/output automata* defined in [97].

external state (such a read/write operation in shared memory, or a non-blocking system call), or performing a blocking system call. Therefore, the application is modelled as a component, interacting with external states and the operating system's kernel by way of notifications. For example, a system call is modelled as a notification sent to the operating system's kernel. The completion of a system call is modelled as the reception of a notification, whose handler models the subsequent instructions.

The operating-system kernel

registers ISRs within the interrupt controller, and, when a device (conceptually another component) raises an interrupt (conceptually sends a notification), the CPU jumps to the address of the corresponding ISR. In the absence of runnable threads, the kernel is idle and only waits to be interrupted.

In hardware

the CPU performs Input/Output (I/O) operation (*e.g.,* `inw` or `outw` instructions) or accesses nonlocal memory (via Direct Memory Access (DMA), or memory that does not reside in a register) by sending requests to the relevant components (*e.g.,* the memory controller or the DMA controller). The CPU effectively stalls until a response is notified back, which can be modelled as the subsequent instructions being the handler to that response.

4.2.2 Timers

While the aforementioned model does not allow to express a time at which a given action shall be performed, software-based generation of a paced packet stream still requires the ability to perform an action after a certain *release-time*. The solution is provided by a component, *a timer*, sending notifications at well-specified times. Any handler executed upon such a notification is therefore known to necessarily complete after a determined time. In particular, when a program is generating a (b, f)-paced stream, each packet transmission must occur upon notification from a timer.

Generalising the software architecture observed in the Linux kernel, timers available to the different components of a commodity server are classified in two categories. Timers from the first category — *clock event devices* in Linux terminology — are components, raising interrupts (periodically or at programmable times), so that software may have a notion of time. Timers from the second category are derived from clocks — *clock sources* in Linux terminology — which must be explicitly requested for the current time, expressed as

a numerical timestamp. Timers from the second category are implemented by probing a clock for the current time, and sending a notification whenever the probed value is posterior to a given time.

Considering timers available to the three types of components described in section 4.2.1:

In hardware

the CPU and chipset have direct access to the clocks originating from one or multiple oscillators on the motherboard. These clocks are used to maintain clock sources such as the Time Stamp Counter (TSC), and clock event devices such as the Local Advanced Programmable Interrupt Controller (LAPIC) timer. The TSC and the LAPIC timer will be used as typical examples of clock sources and clock event devices. This is also without loss of generality, as the following is also applicable to other clock sources and clock event devices (*e.g.,* the High Precision Event Timer (HPET) on x86_64 or the Generic Timer on ARM). The TSC is a register, containing the number of elapsed CPU-cycles since the last reset, and which exhibits an access latency in the order of a few dozen nanoseconds [98]. The LAPIC timer is a device operating asynchronously, and which is to be programmed to raise an interrupt at a programmed time. $t^{TSC,h}$ denotes the time sequence corresponding to TSC-cycles, and $t^{L,h}$ denotes the one corresponding to interrupts from the LAPIC timer.

The operating system kernel

can register an ISR to be executed upon notifications from the LAPIC timer. For the i-th interrupt raised by the LAPIC timer, the time at which the matching ISR starts is denoted $t_i^{L,k}$. That defines a time sequence $t^{L,k}$, different from $t^{L,h}$. Therefore, in kernel-space, software can be specified to be executed *after* times defined by that time sequence. The latency $t_i^{L,k} - t_i^{L,h}$ between the LAPIC i-th interrupt and the start of execution of the matching ISR may vary, intuitively making the time sequence $t^{L,k}$ less regular than $t^{L,h}$.

In kernel-space, software is also able to read the TSC register, compare its value to a given threshold, and execute an action, whenever the read value is larger than the threshold. This effectively enables the kernel to execute a sequence of actions, each starting at times corresponding to a time sequence $t^{TSC,k}$, different from $t^{TSC,h}$, as the former takes into account the latency between the update of the TSC, and the execution of the corresponding action in kernel-space.

In user-space

any system call, specified to suspend a program for a fixed amount of time — such as `nanosleep` — or to schedule a notification at a given time — such as `alarm` — relies on a clock event device (*e.g.,* the LAPIC timer), itself relying on an interrupt. It is serviced in kernel space, the corresponding notification is dispatched to the suspended program, which is finally awoken by the scheduler. Consequently, a user space program can specify actions to be executed at times corresponding to a time sequence $t^{L,u}$ with $t_i^{L,u} - t_i^{L,k}$ being the latency between the beginning of the i-th LAPIC ISR execution, and the time when the user-space program resumes execution.

A user space program can also compare the current time — obtained from a clock source by way of a system call such as `clock_gettime` — to a predefined threshold, so as to start executing an action after a specified time. That allows a user space program to specify actions to be executed according to a time sequence $t^{TSC,u}$ derived from the times at which the TSC is updated[3]. $t^{TSC,u}$ is different from $t^{TSC,k}$ due to the latency between the update of the TSC, and the execution of the specified action in user space.

4.2.3 Timer limitations: drift

In the setup in figure 4.1, if the packet pacing frequency f^p is slightly different from the packet consumption frequency f^c, the buffer B_2 is necessarily subject to overflow or starvation. The cause for this drift between f^c and f^p can be traced to insufficient *timer accuracy*.

The nominal frequencies of hardware clocks are usually given with a tolerance expressed in parts per million (ppm). For example, a 1 MHz clock with a tolerance of 100 ppm has an actual frequency between 999.9 kHz and 1.001 MHz. Because of that inaccuracy, the notion of time given by any timer is never exact. Moreover, due to thermal fluctuations, the frequency of a periodic timer is not stable. Consequently, if a pacer and the CR-PC do not have access to a common source of time, drift will necessarily occur.

4.2.4 Latency

In section 4.2.2, hardware and software layers of a typical commodity server were shown to have access to different timers. Conceptually, and following the terminology of section 4.2.1, timer notifications from hardware to the different software layers are propagated with a certain latency. While most of the literature on real-time operating systems emphasises obtaining

[3]A user space program can also directly probe the TSC without any system call.

latency upper bounds, the definition of pacing given in section 4.1 is insensitive to the addition of any constant delay to the packet transmission time sequence. Consequently, the achievability of packet pacing only depends on obtaining a reduced amplitude between maximal and minimal latency. The sources of those *latency variations* are categorized as:

State-induced

From different initial states, software execution time will differ, yielding differences in latency. At the hardware layer, the micro-architectural state of the CPU — *e.g.,* the state of the caches, or the out-of-order execution pipeline — has an impact on the execution time of an instruction sequence. This variability can be analysed and exploited, as in [99]. Depending on the power-save state of the CPU, the start of execution of an ISR may be subject to a variable latency. At the software layer, the state of the kernel varies from one execution of an ISR to another, which may incur branching in some parts of the code, also yielding variable latency, detectable in user space.

Contention-induced

Latency variations may also be caused by *contention* for a given resource; when two tasks are contending for some resource the latency experienced by one of them varies, depending on whether it has access to the resource before the other one.

Preemption, *i.e.,* suspension of the current execution to process another notification, is a case of contention for a CPU core, and is responsible for latency variations. User-space software may be preempted by higher-priority tasks, scheduled on the same CPU core. A user-space or kernel-space program may be preempted by a non-masked interrupt, because the matching ISR will then need to be executed.

Conversely, the existence of *non-preemptible* sections in the operating system, *i.e.,* sections which cannot be preempted by a high-priority task, is a cause for contention for a CPU core; because a high-priority task is unable to preempt the operating system, it is sporadically delayed, thus, experiences latency variations.

Contention-induced sources are already controllable with Real-Time Operating Systems (RT-OS) such as the PREEMPT-RT patch for the Linux kernel [100] or Xenomai [101]. As much as possible, RT-OSs reduce non-preemptible sections in their kernel, to allow a task, specified as high-priority by the user, to always preempt a low-priority task. For example, the PRE-

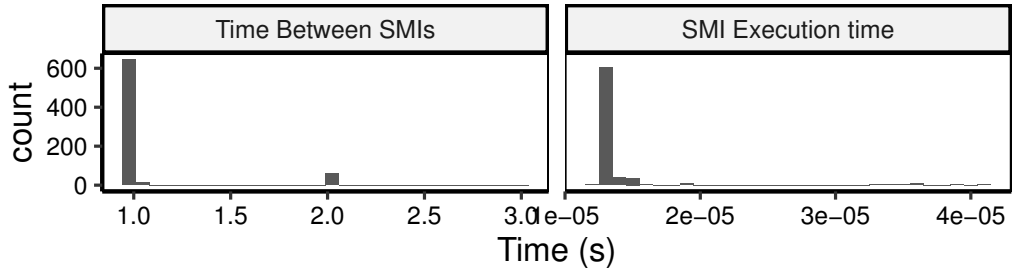


Figure 4.2: Distribution of the time between SMIs and their duration.

EMPT-RT patch replaces the majority of spinlocks from the kernel source code with mutexes, allowing preemption to occur at those points. Consequently, an RT-OS has a lower maximal latency, especially on a highly-loaded system as shown in [100].

However, System Management Interrupts (SMI) are not maskable, may happen sporadically, and do execute transparently to the kernel, *i.e.,* instead of being serviced by an ISR specified by the kernel, the current code execution is suspended to execute firmware code, opaquely to the operating system. SMIs may indistinctly preempt the execution of kernel code or user-space code and are impossible to disable, making them a major source of latency variations, even on an RT-OS. Reducing non-preemptible sections in the operating system's kernel (as performed by the PREEMPT-RT patch for the Linux Kernel) has no impact on SMIs, as those are executed in a context which is transparent to the kernel. A more detailed analysis of the impact of SMIs on system performance is given in [102].

4.2.5 Quantitative analysis of the impact of SMIs

In the following, the pacer of figure 4.1 is a *pure-software* pacer, *i.e.,* is only implemented as a a reactive system, driven by timers, described in section 4.2.2. Due to the latency variations, described in section 4.2.4, the instants at which packets are enqueued into the Network Interface Card (NIC) for transmission, are subject to peak period jitter, itself yielding peak period jitter on the packet transmissions on the wire.

Measurements performed in [103] suggest that SMIs are responsible for a peak period jitter in the order of 20 µs. That is experimentally confirmed by running the `hwlat_detector` tracer, which is part of the Linux tracing subsystem. `hwlat_detector` consists of kernel-space code, specifically designed to measure the impact of SMIs, by polling the TSC in an uninterruptible loop. SMIs are detected when the TSC evolution undergoes irregular jumps,

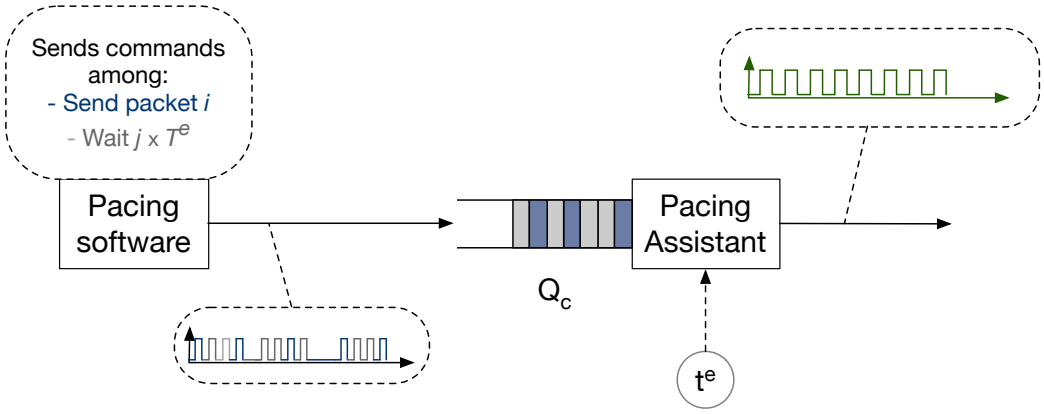


Figure 4.3: Architecture of a PA-based pacer

as those mean that the current CPU core was executing non-kernel code, *i.e.,* it was servicing an SMI. Figure 4.2 shows the obtained results, and confirms that SMIs are happening frequently (every second), and incur a latency ranging from 10 µs to 40 µs.

For a SMPTE-compliant stream received by a typical gateway device, the target frequency f^p of packet transmissions times t^p is in the order of $f^p = 135\,000$ Hz, while the receive buffer size is in the order of $B_2 = 4$. As per property 1, the maximal admissible peak period jitter is $J_p(t^p) = \frac{B_2}{f^p} \approx 30$ µs, which approaches the measured and sporadic additional latency from SMIs. Therefore, in the context of media production, software generation of a (B_2, f^p)-paced stream is potentially unreliable. That result is quantitatively and qualitatively confirmed by the experimental results presented in section 4.6.

4.3 Pacing with a Pacing-Assistant

In sections 4.2.3 and 4.2.5, the peak period jitter obtained by pure software pacing methods was shown to be at least in the order of 40 µs, which is too high for the media production use cases described in the introduction.

As the root cause to that jitter is the unpredictable and unavoidable latency spikes incurred by CPU preemption to service SMIs, a natural solution consists of delegating the time-sensitive tasks of *sending* packets, and of *waiting* between transmissions, to an uninterruptible external system, communicating with the main server through a *command queue*. That approach enables packet-pacing with the required accuracy (contrary to a purely software-based one), while still being implementable with commodity servers and net-

working hardware (see section 4.5), *i.e.,* in a commodity data centre.

4.3.1 Assisted Pacing

That uninterruptible external system is modelled by an abstract component, called a Pacing-Assistant (PA), capable of sending packets spaced by a *precise* amount of time, and not subject to the previously discussed jitter sources, as it is uninterruptible. In other words, that component can send packets, synchronously with a time sequence of negligible ALT-jitter. Defined as an abstraction in this section, a PA is shown to be constructible with general-purpose networking hardware in section 4.5.

Figure 4.3 depicts the architecture of a pacer relying on a PA. A PA is periodically notified by a timer, at times defining a time sequence t^e, with a period T^e, and frequency f^e. The software part of the pacer communicates with the PA by way of *commands*, inserted into a command queue Q_c.

At each t^e-cycle, the PA dequeues a command from Q_c, if one is available. Commands are one of the following categories:

- `wait(n)`: when this command is dequeued by the PA, the next n t^e-cycles are skipped, which is equivalent to waiting $n \times T^e$. Afterwards, the next command in Q_c is dequeued, if available. It is assumed that there is a minimum value n_{min} and a maximum value n_{max} to the admissible values for n. In order to prevent holes in the range of values that PA can wait:
$$n_{max} > 2 \times n_{min} \tag{4.3}$$
- `send(p)`: when this command is dequeued by the PA, the transmission of packet p starts, and lasts for a number of cycles, dependent on the size of p. After the transmission completes, the next command in Q_c is dequeued, if available.

The *cost* of a command is the number of t^e-cycles spent processing it, *i.e.,* the cost of `wait(n)` is n, and the cost of `send(p)` is the number of t^e-cycles necessary to transmit p, denoted by `dur(p)`.

4.3.2 PA-based free-running pacing

As a PA is able to send packets aligned with a time sequence t^e of negligible jitter, a software pacer only needs to enqueue a sequence of `wait` and `send` operations, whose execution (upon dequeueing by the PA) will generate a (B_2, f^c)-paced stream. τ is the target *number of t^e-cycles* between two consecutive packets. For a target period T^c, τ is defined as the ratio $\frac{T^c}{T^e}$.

Algorithm 1: Packet pacing

Primitive : `wait`, `send`, `dur(p)` as defined in section 4.3.1
Primitive : `dequeuePacket`: returns an input packet from B_1
Parameter : n_{min}: minimum value for `wait`
Parameter : n_{max}: maximum value for `wait`
Parameter : τ: target pacing period (in t^e cycles)
Precondition : $n_{max} > 2n_{min}$
Precondition : $\forall p : n_{min} \leq \tau - \texttt{dur(p)}$
Precondition : `dequeuePacket` always returns a packet.

```
1  s ← 0;
2  while True do
3      if s < 1 then
4          p ← dequeuePacket();
5          send(p);
6          s ← s + τ − dur(p);
7      end
8      else if s ≥ n_max + n_min then
9          wait(n_max);
10         s ← s − n_max;
11     end
12     else if s ≤ n_max then
13         wait(⌊s⌋);
14         s ← s − ⌊s⌋;
15     end
16     else
17         wait(n_min);
18         s ← s − n_min;
19     end
20 end
```

Algorithm 1 specifies the sequence of operations to enqueue, so that the PA generates a stream with period T^c.

As used in algorithm 1, the `wait` and `send` primitives **enqueue** the corresponding operation in the command queue Q_c if it is not full, and are blocking otherwise.

Assumption 1. *The command queue Q_c never starves,* i.e., *at each t^e-cycle, the PA is always either processing a `wait` or a `send` command. Also, the commands are reliably enqueued in Q_c.*

Algorithm 1 alternates between issuing a sequence of `wait` commands, and issuing a single `send` command. s is the (possibly fractional) number of t^e cycles between the execution (by the PA) of the last enqueued `wait` command and the execution of the next `send` command to be enqueued.

Thus, a `send` command is enqueued as soon as there is less than a full

t^e-cycle in s. If $s > \lfloor s \rfloor$ (*i.e.,* s is fractional), the remaining fraction of t^e-cycle is accumulated for the next sequence of `wait` operations. Also, as the parameter passed to `wait` must be within $[n_{min}, n_{max}]$, the algorithm slices s appropriately, so that only compliant `wait` calls are performed.

Discussion

Algorithm 1 uses the explicit value τ. If t^e and t^c are neither derived from the same oscillator, nor otherwise synchronized, T^c is not necessarily a rational multiple of T^e. Thus, the quotient $\tau = \frac{T^c}{T^e}$ cannot be meaningfully digitally represented. Also, as stated in section 4.2.3, the nominal values of T^e, and T^c are given with a non-zero tolerance (due, *e.g.,* to the underlying hardware being subject to thermal noise), and thus, cannot be used to compute τ.

Therefore, algorithm 1 is only usable when the timers originating the time sequence t^e and t^c are synchronized, so that τ is known exactly.

4.3.3 PA-based frequency-controlled pacing

When τ is unknown, *frequency-controlled* pacing replaces τ with a *frequency-controller*. A frequency-controller is an external signal F, such that, after the elapse of u t^e-cycles, $F(u)$ estimates the total number of packets transmitted by a CR packet generator of period T^c.

Formally, a frequency-controller is an integer-valued function F, satisfying:

$$\lim_{u \to \infty} \frac{u}{F(u)} = \frac{T^c}{T^e} \tag{4.4}$$

Being an integer-valued function, F is an alternative to an explicit value τ, as it avoids the representation problem described in the discussion of section 4.3.2.

Algorithm 2 uses F to achieve pacing at the target frequency f^c. That algorithm is derived from algorithm 1, but replaces the input parameter τ with a variable τ_{cur}. The core idea driving frequency-controlled pacing, consists of using F to periodically update τ_{cur}. The period of those updates is determined by an input parameter W.

Algorithm 2 maintains a variable y_{now}, containing the total cost of all enqueued operations at a given point of the execution. Under assumption 1, this means that, if an operation is enqueued when $y_{now} = y$, then it will be executed at y-th t^e-cycle by the PA. The value y_{last} of y_{now} at the last update of τ_{cur} is also maintained. Every W t^e-cycles, τ_{cur} is updated to a new value $\frac{y_{now} - y_{last}}{F(y_{now}+W) - F(y_{last}+W)}$. As estimated by F, that value is the average (over a

Algorithm 2: Controlled packet pacing

Primitive : Same as in algorithm 1
Parameter : n_{min}, n_{max} as in algorithm 1
Parameter : F: frequency-controller
Parameter : W: update window
Precondition : $n_{max} > 2n_{min}$
Precondition : $\forall p, \forall a \in \mathbb{N}$:
$n_{min} \leq \frac{W}{F(a+W)-F(a)} - \texttt{dur(p)}$

```
1  s ← 0;
2  τ_cur ← 0;
3  y_last ← −W;
4  y_now ← 0;
5  while True do
6      if s < 1 then
7          if y_now − y_last ≥ W then
8              τ_cur ← (y_now − y_last) / (F(y_now + W) − F(y_last + W));
9              s ← 0;
10             y_last ← y_now;
11         end
12         p ← dequeuePacket();
13         send(p);
14         s ← s + τ_cur − dur(p);
15         y_now ← y_now + dur(p);
16     end
17     else if s ≥ n_max + n_min then
18         wait(n_max);
19         s ← s − n_max;
20         y_now ← y_now + n_max;
21     end
22     else if s ≤ n_max then
23         wait(⌊s⌋);
24         s ← s − ⌊s⌋;
25         y_now ← y_now + ⌊s⌋;
26     end
27     else
28         wait(n_min);
29         s ← s − n_min;
30         y_now ← y_now + n_min;
31     end
32 end
```

duration of $y_{last} - y_{now}$ t^e-cycles) number of t^e-cycles between two packet arrivals of a CR-stream of period T^c. In other words, algorithm 2 performs the same pacing as does algorithm 1, but with a parameter, τ, adjusted periodically to reflect the spacing between packet transmissions estimated

by F.

Also, at a given point of the execution time, frequency-controlled pacing only needs to evaluate F in $F(y_{now} + W)$ and $F(y_{last} + W)$, *i.e.,* two digitally-representable, finite-precision values, making frequency-controlled pacing implementable.

Discussion

Frequency-controlled pacing is designed to avoid drift due to the numerical value $\tau = \frac{T^c}{T^e}$ not being accurately accessible. Algorithm 2 thus implicitly extracts the target pacing rate from an external signal (the frequency-controller), which is therefore required.

4.4 Analysis

This section analytically quantifies the period and ALT-jitter of the packet transmissions obtained when applying the algorithms detailed in section 4.3. As per property 2, those metrics are sufficient to assess whether the transmissions are (B_2, f)-paced. Also, the correctness of those algorithms is verified.

4.4.1 Safety

A program is safe when it can be guaranteed that no program execution will cause an undesirable state to be reached. It is achieved for algorithms 1 and 2 if and only if the arguments passed to the `wait` primitive are within the interval $[n_{min}, n_{max}]$.

The two algorithms assume that, at any time, for any dequeued packet p, and for any value of τ (either given, for algorithm 1, or dynamically computed for algorithm 2) $n_{min} \leq \tau - \texttt{dur(p)}$. The opposite would require the PA to somehow wait less than n_{min} cycles, *i.e.,* that the pacer is requested to pace at a higher-than-its-maximum-frequency. Considering both algorithms at the beginning of the j-th iteration of the `while` loop, the values of the state variables are denoted with index j so that, *e.g.,* the value of s is s_j. $\tau_j = \tau$ for algorithm 1, and $\tau_j = \tau_{cur}$ at the j-th iteration for algorithm 2.

Property 3. *s_j satisfies the following.*

1. *For all $j > 0$, $s_j \in [0, 1) \cup [n_{min}, +\infty)$.*
2. *$s_j \geq 1 \implies n_{min} \leq s_j - s_{j+1} \leq n_{max}$.*

Proof. (See Appendix) □

Algorithms 1 and 2 only call the `wait` primitives when $s_j \geq 1$, with parameter $s_j - s_{j+1}$. The second part of property 3 proves that parameter is always in $[n_{min}, n_{max}]$, hence the safety of the algorithms.

4.4.2 Free-running pacer period

Analysing the state of algorithm 1 at the beginning of the j-th iteration of the main loop, s_j is defined as the value of state variable s, N_j as the total count of enqueued `send` operations, and y_j as the total cost of all enqueued operations. Initially, $s_1 = 0$, $N_1 = 0$ and $y_1 = 0$. At each iteration, s is decremented by the cost of the enqueued PA-operation and incremented by τ if a packet is transmitted, yielding:

$$s_j = \tau N_j - y_j \tag{4.5}$$

For all integers $i > 0$, j_i is defined, so that the j_i-th iteration of the loop enqueues the `send` operation sending the i-th packet. Then, equation (4.5) implies:

$$s_{j_i} = (i-1)\tau - y_{j_i} \tag{4.6}$$

As $s_{j_i} \in [0; 1[$ by construction, and y_{j_i} is an integer number of t^e-cycles, then $y_{j_i} = \lfloor (i-1)\tau \rfloor$. Per assumption 1, y_{j_i} is also the number of elapsed t^e-cycles, when the i-th packet will be transmitted. The time of that transmission is therefore given by:

$$t_i^p = t^e_{\lfloor (i-1)\tau \rfloor} = \tau T^e i + o(i) \tag{4.7}$$

This proves that the period of the paced stream is τT^e.

4.4.3 Frequency-controlled pacer period

A similar analysis is performed for algorithm 2. At the j-th iteration of the main loop, y_j is the total cost of all enqueued operations. j_i is defined such that, the i-th `send` operation is enqueued at the j_i-th iteration of the main loop.

Property 4. *With the previous notations, y_{j_i} satisfies:*

$$y_{j_i} = \left\lfloor W \frac{i - 1 - F(W(k(i)-1))}{F(Wk(i)) - F(W(k(i)-1))} \right\rfloor + W(k(i)-1) \tag{4.8}$$

with $k(i)$ defined as

$$k(i) = \max\{k \geq 1 \mid F(W(k-1)) + 1 \leq i\}$$

Furthermore, τ_{cur} is updated exactly at the iterations of the main loop where the state satisfies $y_{now} - y_{last} = W$.

Proof. (See Appendix) □

By definition of $k(i)$, and by maximality:

$$F(W(k(i)-1)) + 1 \leq i < F(W(k(i))) + 1$$

The second precondition of algorithm 2 yields:

$$F(W(k(i)-1)) + \frac{W}{n_{min}} \geq F(W(k(i)))$$

Combining those two last equations:

$$i - 1 \leq F(W(k(i))) \leq i + \frac{W}{n_{min}} - 1$$

yielding: $F(W(k(i))) = i + o(i)$. Applying in the equation given by property 4 yields:

$$\begin{aligned} y_{j_i} &= O(1) + \frac{W(k(i))}{F(W(k(i)))}(i + o(i)) = \frac{T^c}{T^e} i + o(i) \\ t_i^p &= T^c i + o(i) \end{aligned}$$

Consequently, algorithm 2 results in a packet transmission time sequence of frequency f^c.

4.4.4 ALT-Jitter

The ALT-jitter of the stream, generated by algorithms 1 and 2, can be evaluated for each of the algorithms, as follows

ALT jitter for the free-running pacer

For all i_0 and t_0, and i so that $i \geq i_0$ and $t_i^p \geq t_0$ and from equation (4.7):

$$\begin{aligned} &|T^p(i - i_0) - (t_i^p - t_0)| = |\tau T^e(i - i_0) - (t^e_{\lfloor i\tau \rfloor} - t_0)| \\ \leq &|T^e(\lfloor i\tau \rfloor - \lfloor i_0\tau \rfloor) - (t^e_{\lfloor i\tau \rfloor} - t_0)| + |T^e(\{i\tau\} - \{i_0\tau\})| \\ \leq &\sup_{\substack{j \geq \lfloor i_0\tau \rfloor \\ t_j^e \geq t_0}} |T^e(j - \lfloor i_0\tau \rfloor) - (t_j^e - t_0)| + T^e \end{aligned}$$

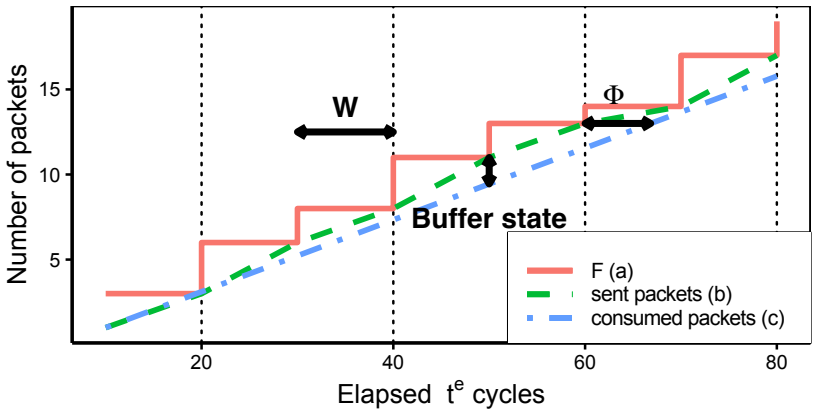


Figure 4.4: Frequency-controlled pacer: F, sent packets , consumed packets by a perfect CR-consumer. Note: the CR consumer on the figure starts at an arbitrary time, *i.e.,* an X-axis offset, and after an arbitrary number of dropped packets, *i.e.,* a Y-axis offset. The ALT jitter is the asymptotic maximum horizontal distance Φ between curve (b) and an optimally-shifted curve (c)

Considering the supremum over all possible i, then the infimum over all i_0 and t_0 yields:

$$J_{ALT}(t^p) \leq J_{ALT}(t^e) + T^e \tag{4.9}$$

The ALT-jitter of the stream, $J_{ALT}(t^p)$, is thus bounded by a component originating from the internal jitter of the PA, and a component bounded by one period T^e. The latter component disappears if T^p is a multiple of T^e, *i.e.,* if $\tau \in \mathbb{N}$.

ALT-jitter for the frequency-controlled pacer

For algorithm 2, t^e is assumed to be perfectly periodic, *i.e.,* that $\forall i$: $t^e_i = T^e i$. This is justified by the fact that the ALT-jitter of t^e is negligible with respect to the one introduced by the variations in τ_{cur}. Figure 4.4 illustrates equation (4.8) by representing (a) the function F, (b) the total number of packets transmitted by the pacer, and (c) the cumulative number of consumed packets by a perfect CR-consumer. On this figure, bounding the ALT jitter consists of asymptotically bounding the horizontal distance Φ between (b) and (c).

$A_W(F)$ and $C_W(F)$ are defined as:

$$\begin{cases} A_W(F) & := T^e \limsup_k |\tau(F(W(k+1)) - F(Wk)) - W| \\ C_W(F) & := T^e \inf_{\substack{k_1 \in \mathbb{N} \\ k_0 \in \mathbb{N}}} \sup_{k > k_0} |\tau F(kW) - kW + k_1 W| \end{cases}$$

$A_W(F)$ is the maximum time interval error when using F to measure a time interval of length W t^e-cycles. $C_W(F)$ measures how accurately τF tracks the number of elapsed t^e cycles on the long term. An upper bound on the ALT-jitter is given by Property 5.

Property 5. *Under the assumption that t^e is perfectly periodic, the ALT-jitter of the stream t^p generated by algorithm 2 is bounded by:*

$$J_{ALT}(t^p) \leq A_W(F) + C_W(F) + T^e \tag{4.10}$$

Proof. (See Appendix) □

This bounds the ALT jitter of t^p into three components:

- $A_W(F)$, which bounds the error $|\tau_{cur} - \tau|$;
- $C_W(F)$, which quantifies how accurately F tracks the number of packets consumed by a perfect consumer, when evaluated at multiples of W; and
- T^e, which bounds the rounding error, since τ_{cur} is not necessarily integer.

4.5 Constructing a Pacing-Assistant and a frequency-controller

Algorithms 1 and 2 rely on a Pacing-Assistant, providing hardware assistance for pacing. Algorithm 2 also requires a frequency-controller. Procedures for building those elements with commodity hardware are described in this section.

4.5.1 Constructing a Pacing-Assistant

This software-based method for implementation of a Pacing-Assistant relies on a commonly-available NIC, and the network infrastructure connected to it. The NIC and network infrastructure must satisfy the following:

- The NIC has a well-defined line rate r, and is able to saturate its output interface at rate r, by consuming packets from a Transmit (TX) queue. The latter consists of an in-memory queue, into which software inserts packets to be sent, and out of which the NIC dequeues packets and actually transmits them on the wire.

- There is a type of packets — *gap* packets — consumed by the NIC at rate r and dropped by some equipment along the network path, before reaching CR-PC, as introduced in [76, 94]. Two possible options for gap packets are (i) packets with a bad Cyclic Redundancy Check (CRC), dropped by any receiver but still effectively consumed by the NIC at rate r, and (ii) IEEE 802.3 Flow Control frames, which should be dropped by any network equipment (*e.g.,* a layer 2 switch) with disabled IEEE 802.3 Flow Control. The maximum IEEE 802.3 frame size (usually at least 1538 bytes) must also be larger than twice the minimum frame size (usually around 64 bytes), so that condition (4.3) from section 4.3.1 holds.

Given that, a PA can be implemented as follows. The PA command queue Q_c is the TX queue of the NIC. The timer of the PA (which events occur at time sequence t^e) is implemented as the *byte-clock* of the NIC, *i.e.,* the timer whose cycles correspond to the transmission of a byte from the TX queue to the wire. The `wait(`u`)` and `send(`p`)` PA commands are implemented as u-sized gap packets and p packets, respectively.

As discussed in section 4.3.2, for an Ethernet NIC whose byte-clock is derived from a local oscillator, and is not synchronized with any external source, drift will occur, effectively limiting the applicability of algorithm 1.

However, the Synchronous Ethernet (SyncE) standard [104, 105] specifies a network architecture, where the Ethernet physical clock of multiple devices — Network Elements (NE) — is derived from a common master by way of a Phase-Locked Loop (PLL), replacing the local free-running oscillator. Consequently, if the NIC used to build the PA relies on SyncE, and the CR-PC relies on the same external time source, then the nominal value T^c/T^e is exact and algorithm 1 can be used.

4.5.2 Constructing a frequency controller: basic version F_b

The frequency-controlled pacer of algorithm 2 relies on a function, F, satisfying equation (4.4), section 4.3.3. If the pacing software is implemented on a system receiving notifications from two timers, at respective time

sequences t^α and t^β, and of respective periods T^c and T^e, then at the u-th notification from the second timer, it can count $F_b(u)$ as the total number of received notifications from the first timer, *i.e.,* $F_b(u)$ is interpreted as the cumulative number of consumed packets by the CR-PC (as t^α is of period T^c) after u t^e-cycles have elapsed (as t^β is of period T^e). Following, and by definition:

$$F_b(u) := \max\left\{i \in \mathbb{N} \;\middle|\; t_i^\alpha \leq t_u^\beta\right\}$$

Also, by definition $t^\alpha_{F_b(u)} \leq t^\beta_u$ and by maximality $t^\alpha_{F_b(u)+1} > t^\beta_u$.

$$\frac{u}{F_b(u)} \geq \frac{u}{F_b(u)} \frac{t^\alpha_{F_b(u)}}{t^\beta_u} \xrightarrow{u\to+\infty} \frac{T^c}{T^e}$$

And

$$\frac{u}{F_b(u)} < \frac{u}{F_b(u)} \frac{t^\alpha_{F_b(u)+1}}{t^\beta_u} \xrightarrow{u\to+\infty} \frac{T^c}{T^e}$$

Thus, F_b satisfies equation 4.4 and is thus a frequency-controller. Consequently, if the system on which algorithm 2 is implemented receives notifications from two timers of period T^c and T^e, a basic version of a suitable frequency controller can be constructed. Methods to implement access to such timers are detailed in the following.

Receiving notifications at t^α

In the abstract setup, depicted in figure 4.1, the CR-PG is transmitting packets with a period T^c. If t^α is defined as the arrivals at the pacer of an *auxiliary* CR packet stream, output directly by the CR-PG, and bypassing best-effort processing, then t^α is a time sequence of period T^c. In the context of media processing, often relying on IP multicast streams, such an auxiliary stream can be implemented by simply replicating (in the network path) the original media stream transmitted by the CR-PG.

Receiving notifications at t^β

At the u-th t^e-cycle, $Q_c(u)$ is the total cost of all the operations in Q_c. With assumption 1 from section 4.3.2, the command queue Q_c is never subject to starvation. In algorithm 2, y_{now} is the cost of all the commands enqueued by the software. With $y_{now}(u)$ as the value of y_{now} at the u-th t^e-cycle, and per assumption 1:

$$Q_c(u) = y_{now}(u) - \max\left\{i \in \mathbb{N} \;\middle|\; t_i^e \leq u\right\}$$

As $Q_c(u)$ is bounded, $\lim_{u\to+\infty} \frac{y_{now}(u)}{u} = f^e$. The time sequence t^β is defined as the sequences of times at which y_{now} increases by 1, *i.e.,* $\forall k : y_{now}(t^\beta_k) = k$. Then:

$$\lim_{k\to+\infty} \frac{t^\beta_k}{k} = \lim_{k\to+\infty} \frac{t^\beta_k}{y_{now}(t^\beta_k)} = T^e$$

The period of t^β is thus T^e, making that time sequence suitable for constructing a frequency-controller F_b. Following that construction, at any time, the value $F_b(y_{now})$ used in algorithm 2 is the number of elapsed t^α cycles and $F_b(y_{last})$, is the value of $F_b(y_{now})$ at the previous iteration.

4.5.3 Constructing F: N_W-regularized version, F_r

Algorithm 2 only evaluates F at multiples of W (see section 4.4), and thus, estimates τ_{cur} at the k-th update as $\frac{W}{F(kW)-F((k-1)W)}$. Consequently, the variations of τ_{cur} will increase with the variability of $F(kW) - F((k-1)W)$. Increasing W has the disadvantage of reducing how often τ_{cur} is updated, making the algorithm more likely to deviate from the targeted period T^c.

Given parameters W and N_W, the N_W-regularized construction of F, F_r is derived from F_b as obtained in section 4.5.2, and is defined as:

$$F_r(t) = \frac{1}{N_W} \sum_{l=0}^{N_W-1} F_b(t - lW)$$

Following that construction, instead of periodically updating τ_{cur} by using the increments $F_b(W + y_{now}) - F_b(W + y_{last})$, algorithm 2 uses a moving average of these increments over the past $N_W \cdot W$ t^e-cycles. That approach allows to smooth the variations of the basic version F_b, but still keeps updating τ_{cur} every W t^e-cycles.

4.5.4 Implementation considerations of algorithms 1 and 2

The validity of the analysis provided in section 4.4, and hence the correctness of the obtained pacing system is conditioned by the assumption 1 in section 4.3.2, *i.e.,* the non-starvation of the command queue Q_c. This is equivalent to the non-starvation of the TX queue of the NIC, which motivates the following implementation choices.

First, algorithms 1, and 2 were implemented as a user-space application, using DPDK for direct access to the NIC. This allows to busy-wait on the state of the TX queue, and to enqueue a packet as soon as possible, hence

maintaining the TX queue full as often as possible. Using DPDK instead of the network stack of the OS kernel also prevents any kernel-originated cross-traffic, which would be injected into the TX queue along with the gap packets and the stream to be paced, and would behave as spurious additional `wait` commands.

Then, to minimise the number of times the OS kernel suspends that user-space application, it is assigned to a specific CPU core, isolated from any other tasks by the `isolcpus` kernel boot option. All hardware interrupts are also rerouted to a CPU core different from that. To avoid any spurious page fault, a call to `mlockall` is performed to guarantee that all the memory used for pacing is locked into physical memory.

As the pacing software thread is alone to be runnable on its assigned CPU core, the Linux kernel is prevented from issuing periodic ticks on that CPU core, as the `nohz_full` kernel boot option is enabled, and the kernel, compiled with the `CONFIG_NO_HZ_FULL` option. The used version of the Linux kernel is the 4.19.3, without the PREEMPT-RT patch, as, in the absence of any concurrent task on the used CPU core, and with all hardware interrupts rerouted to different CPU cores, there is no need for preemption, and hence, no reason to make the kernel more preemptible.

Finally, SMIs are the only remaining cause for the preemption of the pacing software. As stated in section 4.2.5, SMIs may last up to 40 µs, while the time taken by a typical 10 Gbit/s NIC to drain all the 1500-bytes-sized packets from its 512-packets TX queue is in the order of 0.5 ms. As a consequence, SMIs are unlikely to be responsible for the starvation of the TX queue, and, therefore, do not impact assumption 1.

4.6 Experimental Evaluation

The pacing algorithms 1 and 2 are evaluated in experimental scenarios sourced from a media production setup. The experimental setup and methodology are described in section 4.6.1. Qualitative and quantitative results are provided in section 4.6.2, with metrics derived from the peak period jitter and ALT jitter of the paced stream t^p. The frequency-controlled approach is experimentally analysed in section 4.6.3, and an experimental estimation of the two constants $A_W(F)$ and $C_W(F)$, evoked in section 4.4, is also provided.

4.6.1 Setup and methodology

The setup used to evaluate algorithms 1 and 2 is an implementation of the abstraction from figure 4.1, with (i) a CR-PG and a CR-PC, both imple-

Setup	Video output	Video failure type
No Intermediary	Yes	n/a
Linux + iptables	No	Permanent
DPDK forwarding	Yes with failures	Sporadic
pacer (freerun)	Yes with failures	Periodic
pacer (controlled)	Yes	n/a

Table 4.1: Video status of the CR-PC

mented as commodity off the shelf broadcasting pieces of equipment sending and receiving a SMPTE 2022-6 1080i59.94 CR video stream, at a nominative packet rate of $f^c = \frac{4497 \times 30}{1.001} \approx 134\,775.22\,\text{pkts/s}$, (ii) a pacer implemented as software running on an x86_64 Linux server, and implemented as described in section 4.5 (iii) the time sequence t^c (corresponding to packet consumptions by the CR-PC) defined by the signal output by a *tri-level sync generator*, *i.e.,* a piece of broadcasting equipment distributing an out-of-band common clocking signal to CR-PG and CR-PC.

The pacer runs in either frequency-controlled, or free-running, mode, depending on the algorithm under test. The PA is as described in section 4.5.1, and uses IEEE 802.3 flow control frames. The pacing software is implemented as described in section 4.5.4. A 10 Gbit/s network switch ensures the interconnection between the CR-PG, the CR-PC, and the pacer. As described in section 4.5.2, F is constructed by way of an auxiliary stream, implemented as a statically-configured multicast replication of the stream transmitted by the CR-PG. The stream transmitted by the pacer is also replicated to a device — detailed below — for quantitative evaluation.

Unless stated otherwise, the frequency-controlled mode uses a value $W = 1\,\text{s} = 1\,250\,000\,000\,\text{B}$ and an N_W-regularised F with $N_W = 2$. The best-effort processing stage depicted on figure 4.1 is not part of the experimental setup, as the pacer accumulates the packets of the stream of interest into B_1, making it lose all its timing properties. Given this setup, the methodology to obtain quantitative results is described from the perspective of *instrumentation*, and *baseline experiments*.

Instrumentation (detailed in chapter 3)

An accurate evaluation of the jitter of a periodic stream is difficult to obtain using software methods on commodity platforms, for the exact same reasons as those motivating the use of hardware-assisted pacing, *i.e.,* variable latency due to unavoidable causes, *e.g.,* SMIs. Consequently, the measure-

ments are performed using dedicated hardware, implemented on a NetFPGA-SUME programmable card, [75]. The *Open Source Network Tester (OSNT)*[4] [73] for this card appears suitable capturing packets with accurate timestamps. However, a suboptimal Direct Memory Access (DMA) design of OSNT prevents packets and timestamps acquisition, at a rate as high as that of the flow of interest. Consequently, for the purpose of the experiments in this section, the original DMA design in OSNT was replaced by the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) intellectual property core – and, a minimal DPDK driver for acquisition of packets and of timestamps, was implemented.

Timestamp acquisition on this platform is dependent on the accuracy (how valid the nominal frequency is) and jitter of its internal clock. It is assumed (from the datasheet [106] of the Silicon Lab Si5324 oscillator used on the NetFPGA SUME board), that this jitter, being in the order of the nanosecond, is negligible for the results presented in this chapter. However, because of the accuracy limitations of any hardware oscillator — as described in section 4.2.3 — and because no clock drift compensation was implemented, the absolute values of the measured packet transmission timestamps t^p are not independently exploitable, and must be compared to baseline values, obtained as described in the following.

Baseline experiments

The free-running and frequency-controlled PA-based pacers are compared to three baseline experiments:

- No intermediary element exists between the CR-PG and CR-PC. This is expected to show the most regular behaviour.
- The PA-based pacer is replaced with a Linux `iptables`-based setup, redirecting the received stream from the CR-PG to the CR-PC, and using the NetFPGA board for measurement. This is designed to quantify the impact of the Linux networking stack on the periodicity of the stream, thus motivating the need for PA-based pacing, even when the stream undergoes minimal processing.
- The pacer is replaced with a basic DPDK-based forwarder, sending the packets to the CR-PC and the NetFPGA board as fast as possible. This allows to assess whether bypassing the Linux kernel is sufficient to maintain a reliably low-jitter compatible with the CR-PC.

[4]OSNT offers a Verilog/VLSI design for the NetFPGA SUME card.

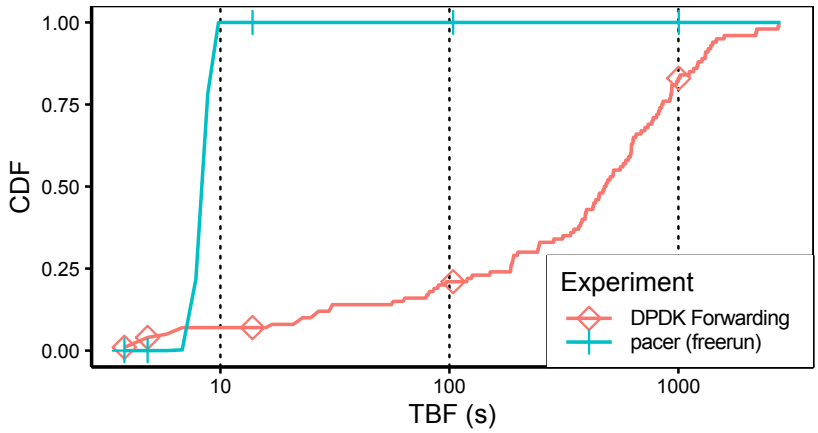


Figure 4.5: Empirical distribution of the Time Between Failures

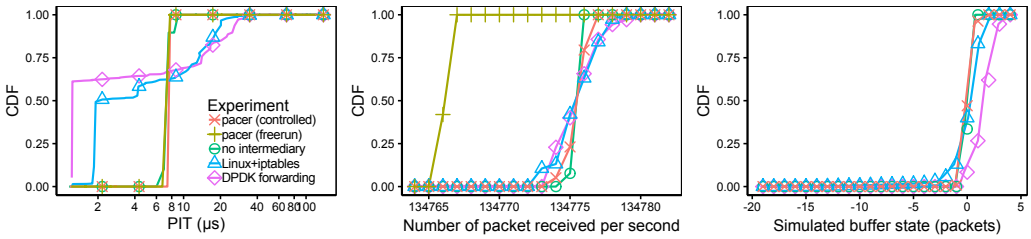


Figure 4.6: Empirical distribution of Packet Inter-arrival Times (left), of packets received during one second (middle), and of a simulated representation of the CR-PC ingress buffer (up to an additive constant).

4.6.2 Results

As the used CR-PC gives no indication about the occupation of its 8-packets buffer, no direct quantitative experimental data can be extracted. However, the CR-PC consumes SMPTE 2022-6 packets to produce video, which gives a qualitative feedback: the presence, or absence, of video. Qualitative and quantitative results are presented hereafter.

Qualitative results: status of CR-PC

Table 4.1 summarizes the state of the video output of CR-PC. In the case where there is no intermediary node between CR-PG and CR-PC, no buffer overflow or starvation is observed, *i.e.,* the video output of CR-PC never stops. Similarly, the frequency-controlled pacer also generates a sufficiently regular stream, so that the video output does not stop.

When going through the Linux network stack, the stream is not suffi-

ciently regular for consumption by the CR-PC, and buffer overflow or starvation are so frequent, that the video never locks, *i.e.,* the receiver never consumes a sufficient number of consecutive SMPTE 2022-6 packets to be able to generate valid video output.

In case (i) of the DPDK-based forwarder, and (ii) of the free-running pacer, video is output by the CR-PC, but interruptions occur, due to overflows or starvations. Qualitatively, the failures seem to happen sporadically in (i), and periodically in (ii). In order to better understand the nature of those failures, the Time Between Failures (TBF) is measured in both cases over twenty-four hours, and its Cumulative Distribution Function (CDF) is shown on figure 4.5.

In case (i), failures are effectively periodic as the CDF is close to a step function, *i.e.,* there is only one value of TBF. Periodic and repeated failures are explained by a mismatch between the period T^p of the output of the free-running pacer, and the period T^c of CR-PC. Because of that mismatch, the buffer occupation increases at a fixed frequency $\Delta f = \frac{1}{T^p} - \frac{1}{T^c}$. As a consequence, after a fixed amount of time proportional to the receive buffer capacity B_2 and the drift Δf, overflow will occur (or starvation if Δf is nonpositive). After that event, a video failure occurs, the CR-PC is reset, and the process repeats with the same drift Δf, hence, failure after the same duration. Consequently, drift-induced failures are periodic.

In case (ii), the CDF shows that the TBF values are spread across a wide range of possible values, *i.e.,* failures are sporadic. This is due to the latency spikes experienced by the DPDK-based forwarding process, triggering violations of property 1, thus starvation or overflow.

Quantitative results: timed captures with the NetFPGA board

For each setup, and corresponding output packet time sequence t^p, and for the i-th transmitted packet, the Packet Inter-arrival Times (PIT), defined as $t^p_{i+1} - t^p_i$, is measured with the NetFPGA board. The statistical deviation of the PIT from the period value T^p is an indicator of how often property 1is violated. Figure 4.6 shows that both versions of the PA-based algorithm produce PIT values which present a step-function-like CDF, *i.e.,* a very reduced peak-period jitter.

Surprisingly, according to the experimental data, the CR-PG does not actually generate a perfectly constant-rate stream, as the PIT distribution is observably different from a Dirac.

The DPDK-based and Linux-based forwarders are *work-conserving* setups, *i.e.,* they do not artificially delay the incoming packets from the CR-PG. As such, the difference between the measured PIT distribution at the output of

both of those setups, and the PIT distribution for the setup with no intermediary, quantifies the distortion introduced by the operating system or the hardware itself. The high-spread observed in both cases in figure 4.6 confirms that, without hardware assistance, accurate pacing is not feasible.

The PIT distributions for algorithms 1 and 2 are indistinguishable on figure 4.6, as the main difference between them is their mean value, *i.e.,* T^p. The CDF of the number of packets received in one-second samples is therefore plotted as well on figure 4.6. This shows that the work-conserving setups and frequency-controlled pacer maintain the target T^p, whereas the free-running version introduces a frequency-drift, which leads to the observed periodic video failures.

The PIT distribution gives fine-grained information quantifying how often the peak period jitter J^p is too high. In order to construct fine-grained information quantifying how often the ALT-jitter is too high, for each i, and for each setup, the value $i - f^c t_i^p$ is computed. This value is the simulated state of a virtual receiver upon reception of the i-th packet of the studied stream, if the receiver starts consuming packets immediately after receiving the first one (*i.e.,* $i_0 = 0$ and $t_0 = 0$, see section 4.1). Figure 4.6 also shows the empirical CDF of this value.

The analysis of figure 4.6 allows to conclude as to why no video was observed in the Linux forwarding case: the graph shows a significant fraction of values far from each other hinting at frequent, large variations of the buffer state, necessarily leading to overflows or starvation. The frequency-controlled pacer leads to a contained buffer occupancy, between -1 and 2, hence $J_{ALT}(t^p) \leq 2$, validating that the built pacer generates a $(4, f^c)$-paced stream as per property 2.

4.6.3 Experimental qualification of F

The bound on the ALT jitter from section 4.4.4 depends on the behaviour of function F, summarised as two constants $A_W(F)$ and $C_W(F)$. As they only depend on the values of F at multiples of W, *i.e.,* values $F(Wk)$ for all integers k, a sample of values $f_{W,k} = F(Wk)$ is experimentally acquired by using the frequency-controlled pacer with parameters $N_W = N_W^0 = 1$ and $W = W^0 \approx 100\,\mu s$. From this base sample, the constants $A_W(F)$ and $C_W(F)$ are estimated for all values of W which are multiple of W^0 and all N_W. Figure 4.7 illustrate these estimates by showing, the sensitivity of these constants both to varying W for a fixed N_W, and to N_W for a fixed W.

This figure gives a conclusive argument for increasing N_W instead of W. A_W — which quantifies the deviation from a perfect CR stream due to the instantaneous frequency-error of the pacer — can be observed to be insens-

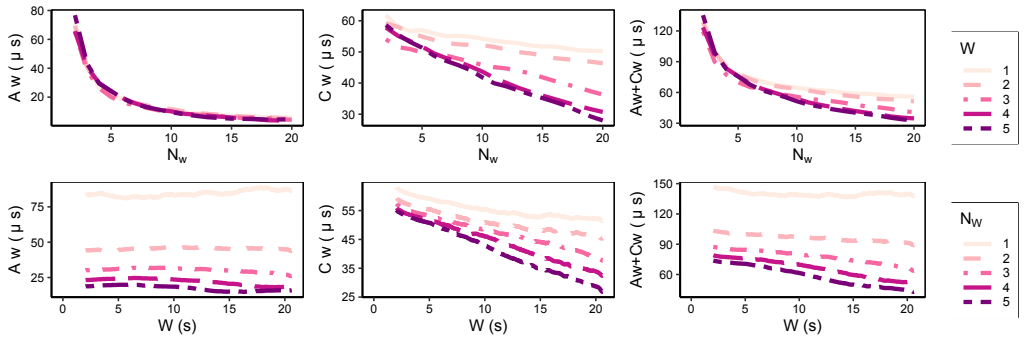


Figure 4.7: Experimental estimation of $A_W(F)$ (maximal error when F is used to measure a time interval of length W), $C_W(F)$ (maximal error when τF is used to track the current time modulo W) and $A_W(F) + C_W(F)$, by varying N_W (regularisation parameter used in the N_W-regularised version of F) and W (sampling period of F).

itive to increasing only W. That result confirms the intuition motivating N_W-regularisation: solely increasing W improves the τ_{cur} estimate, but at the cost of less frequent τ_{cur} updates (as they happen every W). Globally, A_W is therefore not improved by an increased W.

Figure 4.7 also shows that, fixing W and increasing N_W considerably reduces A_W. That result also confirms the intuition: increasing N_W yields a more accurate estimate τ_{cur} but does not change how often τ_{cur} is update.

Figure 4.7 also shows that C_W is similarly sensitive to increasing W or N_W. This is interpreted as an unexpected burst of t^α cycles (see sections 4.5.2) being smoothed by averaging over a longer duration (which is the consequence of both increasing W and N_W). That smoothening improves how well $\tau \times F(Wk)$ tracks the number of t^e-cycles, *i.e.,* Wk.

Finally, $A_W + C_W$, *i.e.,* the bound established in section 4.4, decreases faster when increasing N_W, than it does when increasing W. That gives a conclusive argument for increasing N_W instead of W.

The measured values of $A_W + C_W$ remain higher than the observed ALT jitter for $W = 1\,\mathrm{s}$ and $N_W = 1$. Especially if the ALT jitter were equal to the value of $A_W + C_W$ predicted on figure 4.7, the used CR-consumer would necessarily have starved with an 8-packets buffer. That shows $A_W + C_W + T^e$ to be a fairly conservative bound on the ALT jitter.

4.6.4 Operational perspective

The experimental results show that (B_2, f^c)-pacing is feasible with the frequency-controlled pacing approach. Also, property 5, and the method

used to experimentally evaluate F, yields a practical policy for choosing N_W and W: if a stream is to be paced with a target buffer size B_2, repeating the experiments performed in section 4.6.3 allows to determine which values of N_W and W need to be chosen so that $A_W + C_W + 1 \leq \frac{B_2}{f^c}$.

As per property 5, the stream will then be guaranteed to be (B_2, f^c)-paced.

4.7 Discussion

In a professional broadcasting environment, the loss of more than a single video-frame (due to a single packet loss) per day is unacceptable [107]. In the absence of high-capacity buffers, jitter and drift must be contained. Considering a system with a receive buffer of reduced capacity (in the order of 10 packets), and receiving a packet stream with a frequency of 134775.22 packets per second, the impact of the approach proposed in this chapter is discussed hereafter.

4.7.1 Practical impact of jitter reduction

The model from section 4.1 shows that a consistently small jitter (in the order of a few dozen microseconds) is necessary to enable lossless reception and timely consumption at a CR-PC with small buffers. From an operational perspective, if the experienced jitter is too high, the receiving CR-PC needs to provision a larger buffer. This is not necessarily practically possible, for example, IP-to-SDI gateway devices are usually implemented on FPGAs, with limited and non-evolutive buffering capacity.

Because the proposed pacing system uses commodity servers and general-purpose networking hardware, it is flexible enough to absorb any jitter, and to adapt to any CR-PC.

Finally, the proposed pacing algorithms assume that, whenever a `send` operation is enqueued, the corresponding packet must be available to the pacer. That condition requires that the pacer stores a sufficient number of packets (in the B_1 buffer of figure 4.1), before starting the execution of the chosen pacing algorithm. That initial buffering necessarily introduces some unavoidable delay, depending on the jitter of the input stream. However, this buffering is not specific to pacing; even in the absence thereof, before starting packet consumption, the CR-PC would need buffer the same amount of packets — and, therefore, add the same delay — as would the pacer.

4.7.2 Quantitative impact of drift compensation

The SMPTE stream used in the experimental evaluation has a nominal PIT of $\left(\frac{4497*30\,\mathrm{Hz}}{1.001}\right)^{-1} \approx 7419.761\,32\,\mathrm{ns}$. Introducing, for example, a 0.1 ns error results in a PIT of $7419.76132 - 0.1 = 7419.661\,32\,\mathrm{ns}$, hence an effective packet rate of $\frac{1}{7419.661\,32\,\mathrm{ns}} = 134777.04$ packets per second. That is, a drift of $134777.04 - 134775.22 \approx 1.82$ packets per second.

In the context of professional broadcast, SMPTE streams are unidirectional (allowing, *e.g.,* multicast transmission), and, therefore, no explicit flow control is performed. In that context, if the CR-PC receives 1.82 packets more than consumed every second, a 10 packets receive buffer will overflow after at most $10/1.82 \approx 5.5\,\mathrm{s}$, which is far from the expected reliability of a single frame loss per day. Consequently, achieving pacing for professional broadcast requires the frequency-controlled approach.

4.8 Conclusion

Systems relying on constant-rate packet consumption, and using receivers with small buffers, require transmitted packets to be regularly paced. This chapter shows that such high-accuracy packet pacing can be implemented in software, through designs imposing minimal hardware requirements, captured in the notion of a Pacing-Assistant.

Data processing workflows requiring CR packet streams (such as media processing for broadcasting), and which are traditionally implemented using dedicated hardware, can, using the approach proposed in this chapter, be replaced by *software running on commodity hardware*, and still benefit from a guaranteed *sufficiently regular stream*. Two pacing algorithms were presented and analysed. While the free-running algorithm is only applicable when the Pacing-Assistant and the packet consumer internal clocks are synchronised, the frequency-controlled algorithm has a broader-scope, at the expense of increased operational complexity, arising from the construction and parametrisation of a frequency-controller (as described in section 4.6.4).

An *implementation* of the approach proposed in this chapter has been tested in *real conditions and hardware*, and the viability of software-based packet pacers has been experimentally demonstrated for media-production streams.

From among the conclusions of this chapter, the experiments and analysis presented demonstrate, that the proposed approach is able to bring additional functionalities (pacing regularity and minimisation of buffer occupation), which are not available through standard mechanisms as provided

in general-purpose hardware.

Chapter 5

vMI: Software Architecture for Transparent High-Performance Media Transport

The previous chapters have been qualifying (in chapter 2), measuring (in chapter 3), and overcoming (in chapter 4) any jitter related to software-based packet processing. In particular, chapter 4 demonstrated that commodity servers can be used to transmit media-production packet streams which are compliant with the timing requirement specified by SMPTE 2022-6 and 2110.

The work developed in this chapter goes beyond constant-rate packet transport and analyses software-based media processing in itself, and how it can be implemented on commodity servers in a scalable way. Specifically, this chapter proposes an architecture enabling media processing by way of software running on commodity hardware.

Statement of Purpose

At a glance, this chapter reproduces the media production counterpart of Network Function Virtualisation (NFV) and Service Function Chaining (SFC), which are designed to enable virtualised and software-based network packet-processing.

However, while Virtual Network Functions (VNF) process network packets, the atomic piece of processed information in media-production is a video frame — embedding some audio samples and ancillary data. The existence of such coarse "atoms" is an opportunity to reduce the rate of notifications between the different software elements constituting a media-processing infrastructure. In other words, while a VNF is either notified for every received packet or rely on an (unpredictable) batching heuristic, media-processing

software can be designed so that notifications predictably occur only once per reception of a full video frame.

To this end, this chapter proposes the virtual Media Interface (vMI) software framework, as a way to enable implementing real-time media-processing applications. vMI introduces the abstraction of *media-frames*, *i.e.,* atomic pieces of data for elementary media-processing. By using vMI, a media-processing application only receives media-frames from abstract inputs and transmits media-frames to abstract outputs, vMI being responsible for network operations and packet processing.

Separating packet processing (in vMI), from media-processing (in the application) allows transparent optimisation of the former, *i.e.,* without modifying the media-processing code. Due to the high data-rate of media streams, this chapter shows that the use of **kernel-bypass** and **zero-copy** significantly improves the scalability attainable on some hardware infrastructure.

Finally, the semantic exposed by vMI to media processing applications is similar to the Application Programming Interfaces (APIs) exposed by commercially available SDI-based video extension cards, traditionally used to develop media-processing dedicated appliances. That similarity substantially eases the transition from SDI-based to vMI-based – and, therefore, software-based – media-processing.

Related Work

Kernel bypass is a technique to implement high-performance network applications [108], by providing frameworks wherein packets are treated in user space rather than within the kernel network stack, which reduces the impact of context switches and removes the need for copies between user and kernel space. Kernel bypass frameworks include `netmap` [109] (which reuses the kernel's drivers before mapping packets to userspace), the Data Plane Development Kit (DPDK) [96] (which exposes PCI memory to user space, thus requiring to re-write NIC drivers in userland), or the eXpress Data Path [110]. These frameworks are specifically used to implement VNFs [111–113], virtual switches [82, 114], or user-space Layer-4 stacks [115, 116].

As it will be detailed in section 5.3.2, kernel bypass usually requires a component to perform packet stream demultiplexing, precisely because the network stack of the operating system is bypassed, and, therefore, cannot perform that function anymore. Using a virtual switch for that demultiplexing is, for instance, proposed by the `memif` library [117], which enables user-space applications to connect to the the Vector Packet Processor (VPP) [82] virtual switch and receive packets matched by the Forwarding Information Base (FIB). However, while using the `memif` library imposes a packet copy

between VPP and the application, this chapter proposes a zero-copy architecture, bringing considerable performance benefits.

Service Function Chaining (SFC) [118] consists of pipelining multiple network functions, similarly to media-processing pipelines for media-production. In that context, the performance challenges raised by software-based SFC are studied in [119], while scheduling aspects for VNF chains are explored in [120]. In [121], a disaggregated software-based packet-processing architecture is proposed, wherein each atomic network function can be shared between multiple chains. In the context of media distribution (but not media production), architectures relying on SFC have been introduced, for instance, to assess video quality in cellular networks [122], or to provide video analytics (*e.g.,* motion detection) [123].

Packet-based media processing: While complex video pipelines have been studied and deployed [124], little work has emerged regarding high-performance software processing of SMPTE 2022-6 or 2110 streams. [125] argues in favour of the use of dedicated FPGAs and kernel-bypass stacks to achieve deterministic processing and reach performance and reliability levels of SDI. In [126], a software implementation is introduced that permits capture and reassembly of packet-based video streams, before displaying them locally. Contrary to the work presented in this chapter, [126] does not allow for pipelined video processing. Furthermore, it uses the kernel network stack, and inter-process communication is achieved by writing to and reading from a `pcap` file – thus incurring a large latency (around 10 s) and packet loss rate (around 10%). In [127], an FPGA-based platform to demultiplex SMPTE 2110 streams is proposed, with a microsecond-scale latency. However, the system can only handle one stream at a time, contrary to the work presented in this chapter.

Chapter Outline

The remainder of this chapter is organised as follows. Section 5.2 defines the vMI framework and software architecture. In section 5.3, vMI is shown to be a suitable layer to implement zero-copy and kernel-bypass techniques, and thereby to optimise the transport of media-frames. Section 5.4 describes implementation considerations. Section 5.5 provides an experimental evaluation of vMI. Section 5.6 concludes this chapter.

5.1 Motivation

In this section, the path of a video-frame received by an SDI-based appliance is analysed and, quantitatively compared to the reception of an SMPTE 2022-6 flow on an Ethernet Network Interface Card (NIC).

5.1.1 SDI-based media production: analysis

SDI-based media-processing appliances are proprietary, *i.e.,* their internal implementation is not necessarily fully documented. This section extracts common features from the analysis of open source code, of some well-documented hardware architectures, and of Software Development Kits (SDKs) used by media-processing appliances.

The Video For Linux version 2 (V4L2) framework

To support media devices such as video cameras, digital television tuner extension cards, and video-capture devices devices, the Linux kernel offers an abstraction layer in the form of *the Linux Media Subsystem* [128]. The specific component of that subsystem responsible for transmitting and receiving video frames between hardware devices and user-space applications is the Video For Linux version 2 (V4L2) framework [129].

The Application Programming Interface (API) exposed by V4L2 has the following features:

- V4L2 is **frame-based**: user-space applications which capture video from V4L2 directly receive full video-frames. As a result, the rate at which a user-space application is notified is, at most, the video frame-rate.

- Through the dma-buf sharing API [130], V4L2 allows **descriptor-based video frame manipulation**, enabling an application to receive and send File Descriptors (FD) in place of video buffers. Therefore, if the video data does not need processing, such an FD is directly transferred from an input to an output, with no actual video data-transfer. For example, that can be used to implement software-controlled video-switching, as video only need to be passed from an input to an output.

- V4L2 supports **zero-copy data reception**. Passed as arguments to the `mmap` system call, the aforementioned FD enables direct access to the buffer targeted by the DMA transfer performed upon video frame acquisition.

Even as a piece of software, V4L2 assists hardware-based media-processing, as it can be used with *SDI extension cards*, and hardware processing pipelines spanning over multiple processing devices.

SDI extension cards

Some SDI-based media production appliances consists of commodity servers equipped with compute resources (as Central Processing Units (CPU) and Graphical Processing Units (GPU)) and *SDI extension cards*, providing SDI Input/Outputs (I/O). In many cases, for capturing and generating video, media processing software communicate with those cards through proprietary SDKs such as [131–133].

Among those SDI card, the AJA PCI cards have an open-source driver available [134]. An analysis of that source code shows that the driver integrates into the V4L2 framework, hence generates a maximum of one user-space notification per received video-frame. Furthermore, for each received video-frame, the SDI card raises two interrupts corresponding to the availability of a new video-frame, and the completion of the associated Direct Memory Access transfer. The interrupt rate corresponding to one video stream is, hence, twice the video frame-rate.

Hardware processing pipelines

Another class of media-processing appliances consists of hardware-only solutions, controlled by V4L2. An example thereof, the Zynq-7000 SoC ZC702 Base Targeted Reference Design, a media-processing System on Chip (SoC) integrating a Field Programmable Gateway Array (FPGA) [135], is analysed in the following.

That SoC embeds an ARM processor running a modified Linux kernel [136], and a set of media-processing elements (called *media entities*) implemented in the FPGA. Each media entity has one or multiple I/O channels, and a Linux-based application such as GStreamer [137, 138] is used to chain multiple media entities into a media-processing pipeline. Among those media entities are a video test pattern generator (one output), a video input (one output), a video output (one input), and a Sobel filter (one input, and one output). All these media entities have DMA controllers, with access to a centralized memory space (shared with the ARM processor).

Those media entities have drivers exposing a V4L2 API to user-space applications. A user-space application controls the parameters of the media entities, and, as described in section 5.1.1, is notified upon reception of a video-frame from each of them. By using the dma-buf sharing API, the user-

space application dequeues file descriptors representing video-frames from the output of each media-entity (which is a step of the media-processing pipeline), and enqueues them into the input of the next media-entity in the pipeline. Therefore, the user-space application only sees video-frames as abstract file descriptors and never reads the underlying data, the latter being processed by the hardware media-entities.

5.1.2 Processing high-throughput packet streams for media-production

Two approaches enable applications to receive uncompressed media, transported by packet-streams, as specified by SMPTE 2022-6.

The first consists of hardware-based packet processing, which results in reconstructed video frames being exposed to the software. For example, some video extension cards (similar to the SDI extension cards mentioned in section 5.1.1), have Ethernet sockets instead of SDI coaxial I/Os, and use dedicated hardware to perform packet processing. As a result, that approach would lack flexibility as any evolution in the media-packet format would require hardware changes.

The second approach consists of using NICs, resulting in the software receiving network packets, which would need be processed to extract a media-stream, before being transmitted to the application. Due to its hardware-independence and hence, flexibility, that latter approach is analyzed in this section.

Packet-rate analysis

In the following, a SMPTE 2022-6 stream encapsulating a 1920x1080 interlaced video at $\frac{30}{1.001} \approx 29.97$ frames per second is analyzed. According to [27], a frame is encoded as 1125 lines, each containing 2200 video samples encoded on 20 bits, for a frame size of 6 187 500 B. According to the SMPTE 2022-6 encapsulation [44], each IP packet contains up to 1376 B of video payload, *i.e.,* more than $\frac{6187500}{1376} \approx 4496.72$ packets are required to encode one video frame. Therefore, the packet rate of such a SMPTE 2022-6 stream is $4497 \times \frac{30}{1.001} \approx 134775$ packets per second.

Packet flow

As described in [66], upon arrival at the NIC, a packet is enqueued in a *receive queue*, allocated by the NIC's driver. Then, the NIC raises an interrupt to notify the operating system, which will process the received packet, copy

Table 5.1: Measured interrupt rates upon SMPTE 2022-6 streams reception on a single core.

	With Interrupt Coalescing		Without Interrupt Coalescing	
Stream Count	Interrupt rate	CPU load	Interrupt rate	CPU load
1	7909	0.135	134773	0.484
2	15816	0.289	269536	0.877
3	23724	0.443	367471	0.999
4	31633	0.578	268388	1.00
5	39541	0.716	198792	1.00
6	47449	0.845	122283	1.00

it into a user-space-allocated buffer, and notify the user-space application. This flow thereby incurs at least two context-switches, corresponding to the raised interrupt, and to the notification of the user-space application. At high packet-rates, the overhead incurred by context-switches motivates *batching*, *e.g.,* (i) interrupt coalescing and (ii) polling. By implementing interrupt coalescing, a NIC limits the rate at which it raises interrupts notifying about packet reception, so as to reduce the CPU load. Polling with the New API (NAPI) [139] allows the operating system to mask interrupts from the NIC, and process multiple packets from the receive queue at once.

Interrupt coalescing and NAPI-based polling use heuristics to determine optimal batch sizes, ignoring the application-level batches occurring when full video frames — *e.g.,* 4497 packets for streams such as described in section 5.1.2 — are received. Table 5.1 depicts the interrupt count and CPU load when a single CPU core is exposed to a varying number of SMPTE 2022-6 streams. Even with interrupt coalescing, the interrupt rate observed is multiple orders of magnitude higher than the video framerate, as opposed to the interrupt rate resulting from video reception on SDI extension cards, as described sections 5.1.1 and 5.1.1.

A high-interrupt rate can artificially degrade the performance of packet-processing, as described in [140]. Moreover, as the user-space application is notified of incoming packets, that high measured interrupt-rate is responsible for a high context-switch rate, which is detrimental to performance. That issue is amplified when pipelining multiple media-processing applications on one server, if packet-based transport is also used between successive stages.

Towards a virtual Media Interfaces (vMI)

The limitations described in section 5.1.2 motivate an architecture providing an abstraction layer responsible for the transport of media-frames. In particular, that abstraction layer should mediate all video-frame-based (or more generally *media-frame-based*) communication between the media-processing application, and commodity NICs. That layer would provide a *virtual Media Interface (vMI)*, mimicking the API offered by software frameworks interacting with SDI-based processing hardware. In particular, vMI targets:
Support for multiple transports: Similarly to unified frameworks such as V4L2, which are compatible with a variety of video devices, vMI aims at providing multiple types of media-transports, packet-based, or not.
Low notification overhead: An application relying on vMI should only perform one sending call to transmit a media-frame, and only be notified once per received media-frame. Furthermore, depending on the underlying transport, vMI should be designed so as to minimize notifications from the operating system or other processes.
An endpoint-agnostic API: While a packet-receiving/transmitting application must be aware of the packets' origin/destination, vMI should only expose opaque inputs/outputs. This allows media-frame reception/transmission, independently from the identity of the remote transmitter/receiver. Such an endpoint-agnostic API is natural in a media-production environment, as SDI-cabling is part of the infrastructure, not of media-processing.
Buffer-sharing and zero-copy: One of the underlying motivation to softwarization is gained flexibility, in particular through *modularity*. By pursuing *interprocess* media-frame sharing and zero-copy media-frame transmission (similarly to the V4L2 framework described in section 5.1.1), vMI prevents the performance degradation described in section 5.1.2 upon application chaining. That enables the construction of a disaggregated media-processing pipeline, *i.e.,* a modular chain of user-space applications, each performing an elementary operation on the media-stream.

5.2 Overview of the vMI framework

In this section, the vMI framework is described, and its use to implement media-processing application is illustrated.

5.2.1 Main Concepts

To provide media-processing applications with an abstract interface, hiding *how* media is transported, vMI introduces the concepts illustrated in

figure 5.1 and defined in the following.
vMI Frames are objects, manipulated by a media-processing application, and represent media-frames. They are composed of *media-data*, *e.g.,* the pixel values of a video frame, and *vMI metadata*, *e.g.,* a media sampling timestamp, a media-frame sequence number, the type of media-data (audio or video), etc.
A vMI Module is the software equivalent of an SDI extension card. A vMI module is instantiated by a media-processing application and is configured with a set of *vMI inputs and outputs*, equivalents of I/O connectors on an SDI extension card, and a *vMI callback*. Each vMI input and output has its own configuration, embedded in the configuration of the vMI module. The vMI callback is a routine, specific to the application, and associated with the vMI module. Upon reception of any complete media-frame, that routine is executed, similarly to an interrupt service routine called when an SDI card receives a full video-frame. Therefore, the callback routine of a vMI module with i inputs, each receiving a stream at a frame-rate f, would be called at a rate $f \times i$.
vMI Inputs/Outputs (I/O) are endpoints for the reception and transmission of vMI frames from the infrastructure. They are never directly manipulated by the application. They have a type, depending on the underlying channel used to receive or transmit vMI frames. Some examples of vMI I/O types are listed in table 5.2, along with their configuration. For example, a vMI input of type SMPTE 2022-6 is used to receive an SMPTE 2022-6 stream from a network interface, decode it, and transform the received video frames into vMI frames usable by the application.

5.2.2 The Flow of a vMI frame

Figure 5.1 depicts the flow of a vMI frame through a media-processing application. A vMI input receives data from a communication channel (①) and builds a vMI frame. Then, the vMI callback routine is executed (②) with arguments including an opaque value, identifying the received frame, and an index identifying the corresponding input. That application-dependent routine triggers actual processing on the frame (*e.g.,* image processing). The vMI module is then requested to transmit the modified vMI frame (③) to the infrastructure, via a vMI output (identified by its index). Finally, the corresponding vMI output performs that transmission (④).

Table 5.2: Examples of vMI I/O types

Type	Description	Configuration
SMPTE 2022-6 Input	On the network interface with IP address *interfaceAddress*, subscribe to the multicast group *mgrp*, listen to UDP port *port* to receive an SMPTE 2022-6 stream and decode it.	mcastgroup=*mgrp*, ip=*interfaceAddress*, port=*port*
SMPTE 2022-6 Output	Encode vMI frames into an SMPTE 2022-6 stream, and transmit it to the multicast group *mgrp*, on UDP port *port*, through the network interface with IP address *interfaceAddress*.	mcastgroup=*mgrp*, ip=*interfaceAddress*, port=*port*
Shared Memory Input	Receive vMI frames from a shared memory segment identified by *id*, whenever notified on UDP port *id*.	control=*id*
Shared Memory Output	Copy vMI frames into a shared memory segment identified by *id*, and sends a notification on UDP port *id*.	control=*id*

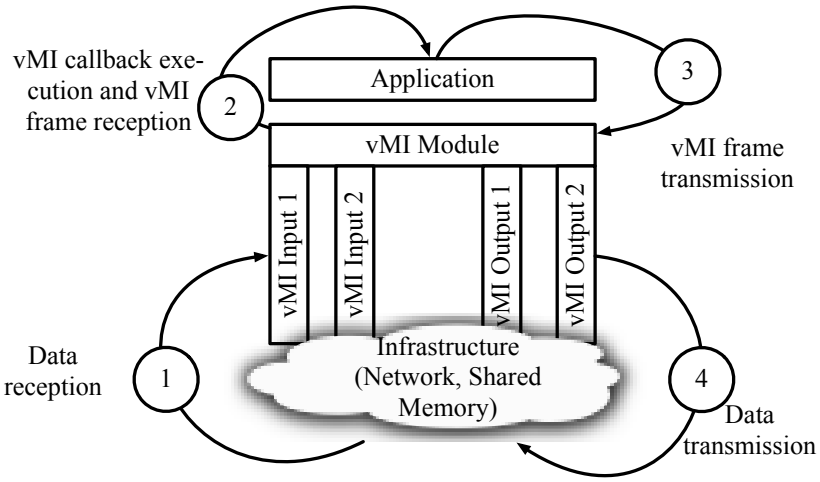


Figure 5.1: Flow of a vMI frame through a media-processing application

5.2.3 Disaggregated media-processing

A *media-processing pipeline* is a sequence of vMI-based applications, each performing some elementary media-processing. Compared to monolithic media-processing (*e.g.,* as performed by dedicated, SDI-based, appliances), such a *disaggregated media-processing* architecture allows to seamlessly distribute compute-intensive media-processing, either over multiple CPUs on the same server or over multiple physical servers.

As depicted in figure 5.2, applications in a media-processing pipeline must be configured so as to ensure the connection of the vMI output of each application, to the vMI input of the next in the pipeline. As shown in table 5.2, such a connection can be implemented over the network (*e.g.,* with a pair of SMPTE 2022-6 I/Os) or over Inter-Process Communication (IPC) channels provided by the operating system (*e.g.,* with a pair of Shared Memory I/Os). vMI enables seamless switching between those different transport types, as it only requires the reconfiguration of the vMI modules, hence additional flexibility in the construction of media-processing pipelines.

5.3 High-performance vMI frame transport

This section describes how transport of vMI frames can be optimised by way of vMI I/Os using *kernel-bypass* and *zero-copy*. This allows implementing a media-processing pipeline as described in section 5.2, with minimal notification overhead, and minimal data copies.

5.3.1 Interprocess vMI frame sharing

The shared memory vMI I/Os in table 5.2 impose a memory copy when a vMI frame is sent through a vMI output. This is because those I/Os only share memory segments between one vMI input and one vMI output. For example, in a media processing pipeline with three applications A_1, A_2 and A_3, and using two shared memory segments $S_{1\to2}$ (connecting the output of A_1 to the input of A_2) and $S_{2\to3}$ (connecting the output of A_2 to the input of A_3), A_3 does not have access to $S_{1\to2}$. Thus a vMI frame copy is necessary between $S_{1\to2}$ and $S_{2\to3}$. A pipeline comprising n applications would, therefore, require at least $n-2$ copies.

To enable all processes (hosted on the same server) in a media-processing pipeline to operate with a minimal number of copies, they should, therefore, all have access to a shared memory region, used to allocate all the vMI frames. To that end, vMI uses the multiprocessing capabilities of the Data-Plane Development Kit (DPDK) [96].

DPDK-based memory sharing

DPDK implements multi-processing capabilities so that multiple independent processes, whose memory space is theoretically isolated from each other, can collaborate to process packets. To that end, DPDK provides a memory allocator which uses *hugepages* [141], *i.e.,* operating-system-provided memory regions, identified by a file descriptor, shareable across process, and necessarily backed by contiguous physical memory. How DPDK allows multiple processes to access the same memory is outlined in the steps below.
Step 1: A first process, A_M, is started, and initialises the DPDK library in *primary* mode. A_M will map some hugepages into virtual memory (by way of the `mmap` system call), which are then used to initialise DPDK's internal structures. The mappings between each hugepage and the corresponding virtual memory address are finally recorded in a specific file f.
Step 2: Another process A_1 is started, and initialises the DPDK library in *secondary* mode. Each hugepage-to-virtual-memory address mapping, recorded in f, is tentatively reproduced by A_1, by using the `mmap` system call. [1]
Step 3: Other secondary processes $A_2, A_3, ...$ go through step 2 and therefore can all see the same hugepages, mapped at the same virtual addresses, as if they were part of the same memory space.

A vMI module allows specifying whether vMI must initialise DPDK, in

[1] Because of Address Space Layout Randomisation (ALSR) [142], this can fail as the virtual memory addresses used in one of the mapping can collide with the heap, stack or code of A_1

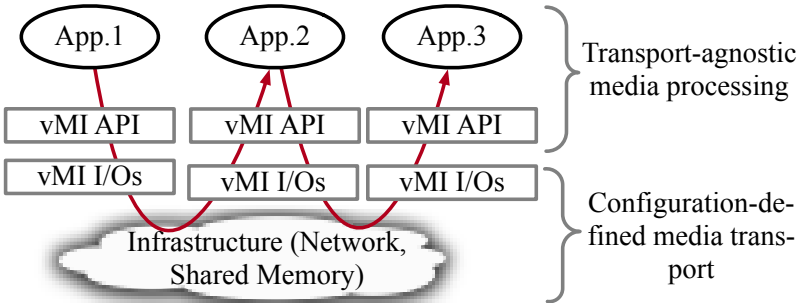


Figure 5.2: A vMI-based media-processing pipeline

which case, all vMI frames will be allocated by using the DPDK-provided allocator and, hence, are shareable among multiple processes in the same media-processing pipeline.

DPDK-based message-passing

DPDK also provides synchronisation by way of a message API, allowing a primary or secondary process to register a message-handler callback, executed each time a certain *message* (identified by a string) is received. If a primary process A_M sends a message m, all secondary processes $A_1, A_2...$ will execute their message-handler callback for m, if any. If a secondary process A_1 sends a message m, only the primary process A_M will execute the message-handler callback for m. By using that message API, each process in a media-processing pipeline can signal to the next that a vMI frame is available for processing.

Zero-copy vMI shared memory I/Os

The following steps describe the use of the DPDK features introduced in sections 5.3.1 and 5.3.1 to build a zero-copy media-processing pipeline. To that end, **zero-copy vMI shared memory I/Os** are introduced, in addition to the I/Os described in table 5.2.

1. First, a unique vMI master process A_M initialises DPDK as a primary process. A_M is not a media-processing application, but is part of the infrastructure, with as sole responsibility, the role of a hub for messages between media processing applications. This is because secondary-to-secondary messages are not supported, per section 5.3.1.

2. Then vMI-based media-processing applications $A_1, A_2, ...$ are started as secondary processes and configured to use DPDK allocated memory, as

described in section 5.3.1. These processes have zero-copy vMI inputs and/or zero-copy vMI outputs, each configured with a *queue identifier* and a *notification identifier*. Each input allocates (in shared memory) a queue of pointers to vMI frames, recorded in a shared-memory structure under the same name as the *queue identifier*. This queue will be used by the input to receive vMI frames. Each input also registers a message-handler callback for the message named as the *notification identifier*. This message-handler callback is executed whenever a new vMI frame is available in the queue.

Once all vMI I/Os are initialised, a flow of frames can follow a zero-copy path across multiple media-processing applications $A_1, A_2, ...$ Each application A_i in the pipeline has a zero-copy vMI output directly connected to a zero-copy vMI input of A_{i+1}. Those vMI I/Os are both configured with a queue identifier $Q_{i,i+1}$, and with a notification identifier $N_{i,i+1}$.

First, a vMI frame is allocated, processed in A_1, and enqueued for transmission on a zero-copy vMI output. vMI then inserts the address of that vMI frame in the queue, identified $Q_{1,2}$, and sends the notification $N_{1,2}$, received by the vMI master, which, as a hub, repeats it to all secondary processes. Due to the initialisation of its zero-copy vMI input, A_2 executes the message-handler callback for $N_{1,2}$, signalling the availability, in queue $Q_{1,2}$, of a vMI frame address (also valid in A_2, per section 5.3.1), which is finally dequeued, and passed to the media-processing part of A_2. The latter performs some in-place processing, and similarly sends out the vMI frame to A_3, and so on.

5.3.2 Kernel-bypass networking

Kernel-bypass networking consists of moving packet processing from the kernel network-stack to a user-space process. In that setting, the kernel is therefore not able to perform IP- or port-based flow demultiplexing upon packet reception, and applications using kernel-bypass necessarily follow one of three models:

No-demultiplexing implies that each network interface is fully owned by a single application, *i.e.,* packets received on that interface are processed by that application. That is simple to deploy, as the set of network interfaces used by each application are disjoint, thus little interference is possible. However, given that a media-stream can have a data-rate as low as 1.5 Gbit/s, while the line-rate of a typical server-grade NIC is 10 Gbit/s, that model is likely to yield network capacity underutilisation: a media-processing application receiving one single stream would still own a full network interface, hence the waste of some 8.5 Gbit/s worth of capacity.

Hardware-assisted demultiplexing uses the fact that some NICs can match received packets against a set of rules, specified by the driver: each packet matching a rule is inserted into a specified Receive (RX) queue. A NIC implementing that feature can perform demultiplexing if, at the software layer, there is a way to assign a queue to a given user-space process. Remote Direct Memory Access (RDMA) [143] is one hardware-assisted kernel-bypass technique, allowing received packets matching specific rules to be directly delivered to the memory of a user-space process, *i.e.,* RDMA-capable NICs include hardware-assisted demultiplexing. NICs implementing Single-Root Input Output/Virtualisation (SR-IOV) [144] expose a set of Virtual Functions (VFs) to the operating system, each VF appearing as an independent interface. Demultiplexing can, therefore, be achieved by configuring each VF to receive packets matching an application-specific set of rules, and then assigning it to the corresponding application. Contrary to the absence of demultiplexing, that model allows to fully use the capacity of a NIC, at the cost of increased operational complexity, and hardware-independence, as each NIC has its own demultiplexing capabilities, and must be configured independently.
Software-mediated demultiplexing implies that all NICs are owned by a *virtual switch*, *i.e.,* a user-space process receiving all the incoming packets, and taking all demultiplexing decisions. Replacing the operating system's network stack by a user-space application, the virtual switch is, therefore, required to provide high-performance packet-processing. Once a packet is demultiplexed by the virtual switch, an IPC mechanism is used to pass the packet to the relevant application. That is hardware-independent, but requires an additional software component (the virtual switch), increasing operational complexity and resource usage.

To maximise hardware-independence (one of the goals of the softwarisation of media production) vMI relies on the software-mediated demultiplexing model, and uses VPP as a virtual switch. No packet-copy is performed when packets are received by VPP and are passed to a media-processing application through vMI, saving memory bandwidth and reducing latency.

Flow establishment

The SMPTE 2022-6 vMI input from table 5.2 is extended with a "VPP" mode to enable kernel bypass. In that mode, the vMI input initialises by connecting to a running instance of VPP (through the VPP API) and passing all the relevant networking parameters (multicast group, port, interface) of the media stream to receive. Similar to DPDK's memory sharing (section 5.3.1), the vMI input maps the memory used by VPP, so that all the packet vir-

Algorithm 3: Batched Polling algorithm

Parameter : V (batch size)
Primitive : `getRXQueueSize()`: get the current number of enqueued packets
Primitive : `dequeueFromRXQueue(n)`: dequeue n packet addresses from the packet RX queue and return them in an array
Primitive : `processPkts(a)`: decode the packets in array and extract media data
Primitive : `yieldScheduler()`: yields so that the operating system can schedule other threads

```
1  pendingArray ← array of size V;
2  while True do
3  |  available ← getRXQueueSize();
4  |  if available ≥ V then
5  |  |  pendingArray ← dequeueFromRXQueue(V);
6  |  |  processPkts(pendingArray) ;
7  |  end
8  |  else
9  |  |  yieldScheduler() ;
10 |  end
11 end
```

tual memory addresses which are seen by VPP, become valid to the media processing application. Then two queues are created: the *packet RX queue*, and the *packet recycling queue*, whose uses are described hereafter. Finally, an entry is created in VPP's Forwarding Information Base (FIB), so that received packet matching the vMI input configuration (multicast group, port, and interface) are passed to the application.

Packet receive flow

VPP constantly polls the state of all the RX queues of all the NICs to which it has access. When a NIC receives a packet, it is transferred using DMA at a location, pre-allocated by VPP. It is then processed, and matched against the FIB. If it matches a rule corresponding to a vMI input, the address of that packet is enqueued in the corresponding packet RX queue, located in a memory area shared among all vMI-based applications.

The application, to which the vMI input belongs, is actively polling the vMI input packet RX queue. Arriving packets are dequeued, reassembled, and decoded as a vMI frame, before being passed to the media-processing application. At that point, the packet buffers are no longer needed, and all

relevant media-data has been extracted by vMI.

Buffer recycling flow

Unlike the zero-copy flow for full vMI frames, described in section 5.3.1, the buffer transmitted from VPP to a media-processing application is not allocated by the vMI framework, but by VPP. Thus, it must also be deallocated by VPP. To that end, when a received packet is no longer needed by the vMI input, its address is inserted into the packet recycling queue. Whenever VPP sees an address in the packet recycling queue, it deallocates the corresponding buffer.

Performance considerations

Unlike vMI frames, which are transmitted at the *frame-rate* of the video (between 25 and 60 frames per second), packets are enqueued and dequeued from the packet RX queue at the *packet-rate* of the stream (in the order of 100000 packets per second). Such a high rate prevents VPP from explicitly notifying the application (as described in section 5.3.1 for vMI frames), which is, therefore, required to *poll* the packet RX queue. As it is shared between the VPP thread performing packet processing and the application thread consuming the packets, frequent enqueue and dequeue operations are a cause for performance degradation. That is mainly due to *locking*, and *cache-line invalidation*.

Locking occurs in a multi-producer or multi-consumer scenario with multiple threads concurrently enqueuing/dequeuing data into/from a shared memory queue. It occurs when using, *e.g.,* atomic Compare-and-SWap (CSW) instructions to increment/decrement pointers to the next-available or last-used slot in the queue, as described in [145]. Locking is avoided in vMI by imposing that each packet RX queue has a single producer and a single consumer.

As documented in the MESI cache-coherency protocol [146], cache-line invalidation necessarily occurs as the consumer thread updates the last-used slot, which needs to be read by the producing thread to determine if there is enough room to enqueue a new element or not. That is avoided by performing a single dequeue operation for multiple packets at once. That approach is detailed in algorithm 3, describing how packets from the packet RX queues are consumed by vMI.

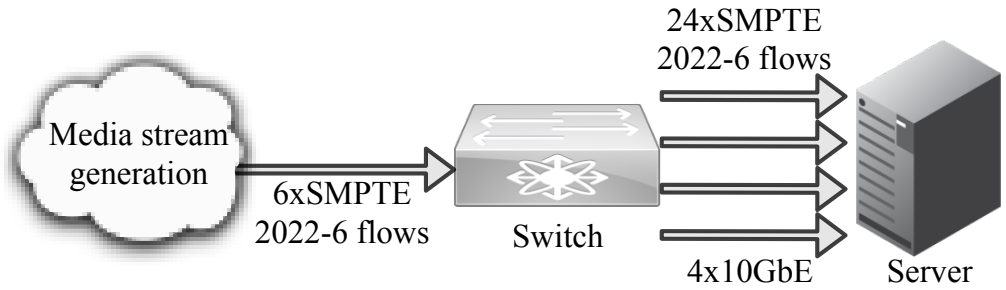


Figure 5.3: Hardware setup used in the conducted experiments

5.4 Implementation

The vMI framework as presented in section 5.2 has been implemented and has been extended with the high-performance vMI I/Os presented in section 5.3, *i.e.,* zero-copy shared memory I/Os as well as VPP-based inputs. Moreover, both the vMI master and virtual switch features of sections 5.3.1 and 5.3.2 were implemented in a single VPP plugin, effectively merging them into a single process, supporting the infrastructure necessary to high-performance media-transport.

5.5 Evaluation

The high-performance media transport methods developed in section 5.3 are, first, microbenchmarked, to determine the impact of VPP-based media transport on resource usage and scalability. Then, a realistic media-processing pipeline is implemented, and the impact of memory sharing on vMI frame transport is quantified.

5.5.1 Experimental methodology

Figure 5.3 depicts the hardware setup used. A media stream generator, composed of an SDI player and an SDI-to-SMPTE 2022-6 converter, produces a total of six multicast SMPTE 2022-6 streams. Each one of those is a 1080i59.94 video stream, transported over a UDP/IP stream amounting to 134775 packets per second, and 1.5 Gbit/s. The generator is connected to a non-blocking 10 Gbit/s packet switch. Software media-processing is implemented on an x86 server with two Intel Xeon E5-2690 v4 CPUs, totalising 28 cores, 128 GB of quad-channel 2400MHz DDR4 RAM, and an Intel X710 NIC with four 10 Gbit/s ports all connected to the packet switch. That setup

allows each of the six multicast streams to be replicated onto the four ports of the NIC, allowing reception by the server of up to 24 streams.

The server runs a Linux 4.19.3 kernel, configured in full dynamic tick mode[2], with 24 CPU cores isolated from the scheduler to reduce interference from threads not involved in the experiments. Also, interrupt requests from devices, other than the NIC, are routed to one of the four remaining CPU cores.

Parameters and Metrics

The experiments consist of receiving and processing N media-streams on the server. The system scalability is evaluated by increasing N, and evaluating the reliability of each processing pipeline, using two metrics, reported every second.

Losses Per Second (LPS): Due to buffer overflows, packets may be dropped at some point in time between arrival at the NIC and consumption by vMI. SMPTE 2022-6 streams are not resilient to packet drops, making such events critical errors triggering video interruption. The LPS metric is generated by evaluating, over consecutive one-second intervals, whether a loss happened, and computing the ratio of the number of intervals with losses, over the total number of intervals.

Frames Per Second (FPS): Due to insufficient compute resources, a media-processing pipeline may not be able to process vMI frames at the nominal video frame rate, hence a drop in the number of processed frames per second.

Memory Bandwidth (MEMBW): The zero-copy optimisations presented in section 5.3 aims at reducing the impact of media streams on the memory bus usage, quantified by sampling hardware performance counters embedded in the memory controller.

CPU core allocation strategy

Between an SMPTE 2022-6 packet arrival at the NIC, and the completion of the associated media processing pipeline, these compute-intensive *tasks* are performed, all in different threads: First, the packet undergoes *network processing* either in kernel-space or, if VPP is used, in user-space. Then, *application reception* occurs, either through a call to the socket API or, if VPP is used, by packet RX queue polling. Finally, the packet is *parsed* to build a vMI frame.

During the experiments presented hereafter, available CPU cores are partitioned into *task groups* $T_1, T_2, ..., T_n$, each associated to one of the aforemen-

[2] `https://lwn.net/Articles/549580/`

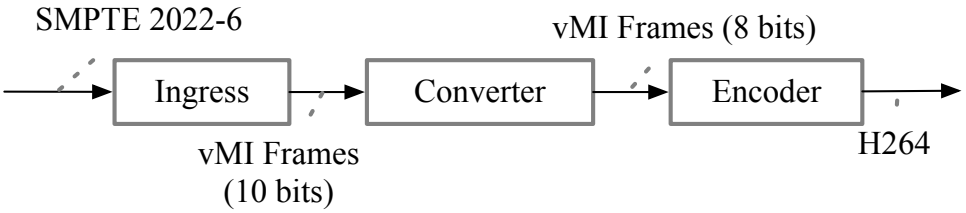


Figure 5.4: Logical view of the three-stage pipeline used in the conducted experimental evaluation. The ingress receives a SMPTE 2022-6 stream, and decodes it into a sequence of vMI frames containing containing 10 bits video, which is then transformed into 10 bits video by the converter, and finally encoded into an H264 stream.

tioned tasks (network processing, application reception, and parsing). The presented results are parameterised by the number C_i of CPU cores allocated to each task group T_i. To avoid an intractable, exhaustive, parameter-space exploration, CPU cores are allocated to tasks by iteratively increasing N, and determining the limiting task group T_l. A CPU core is then tentatively moved from one of the other task groups T_j to the limiting task group, such that C_l is incremented and C_j is decremented. If doing so is possible without making T_j a limiting taskgroup, that process is iterated. Otherwise, the obtained allocation is considered a local optimum.

5.5.2 Microbenchmarks

The impact of VPP-based kernel bypass is evaluated on a single-stage media-processing pipeline, comprising an application configured with an SMPTE 2022-6 input, using either VPP or the kernel network stack, and simply dropping the received vMI frames. Microbenchmarking only uses 12 out of the 14 CPU cores on the first CPU, as it also controls the PCI Express bus connected to the NIC. The two remaining cores are used for tasks unrelated to media-processing, such as gathering experimental results.

Task groups description

Packets are processed differently when using the kernel network-stack, or when using VPP. In the former case, packet-processing is interrupt-driven and is performed by the CPU core which received the corresponding Interrupt ReQuest (IRQ). In the following, CPU cores assigned to receiving those IRQs are called *IRQ Cores*. In the latter case, packet-processing is

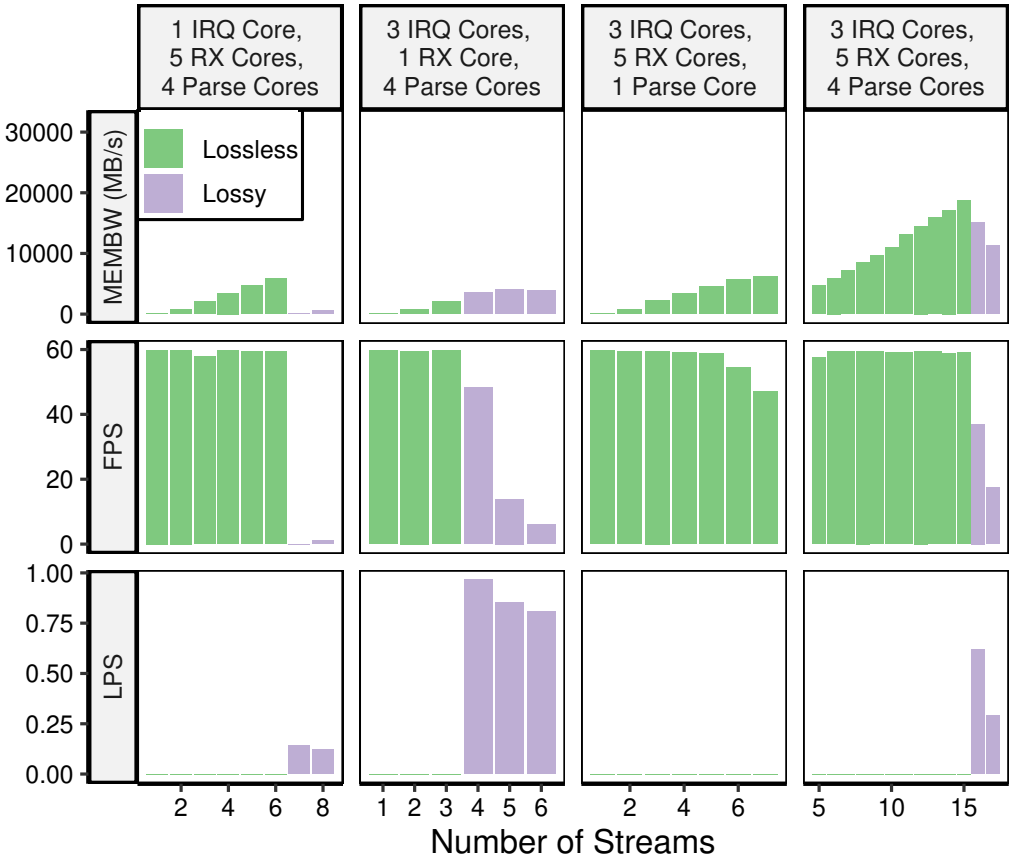


Figure 5.5: vMI performance using the kernel network stack, with an application only receiving vMI frames. The optimal allocation is represented in the last column, as the total number of available cores is 12 = 3 IRQ Cores + 5 RX Cores + 4 Parse Cores. Because they represent Losses Per Second (LPS), the graphs of the third row show no data points in lossless configurations.

performed by VPP, which at least need two CPU cores[3]. The number of CPU cores affected to VPP was never the limiting factor in the conducted experiments, and hence, is not part of the parameter-space considered. CPU cores performing application-reception are referred to as *RX Cores*, the ones performing SMPTE 2022-6 parsing and decoding are called *Parse Cores*.

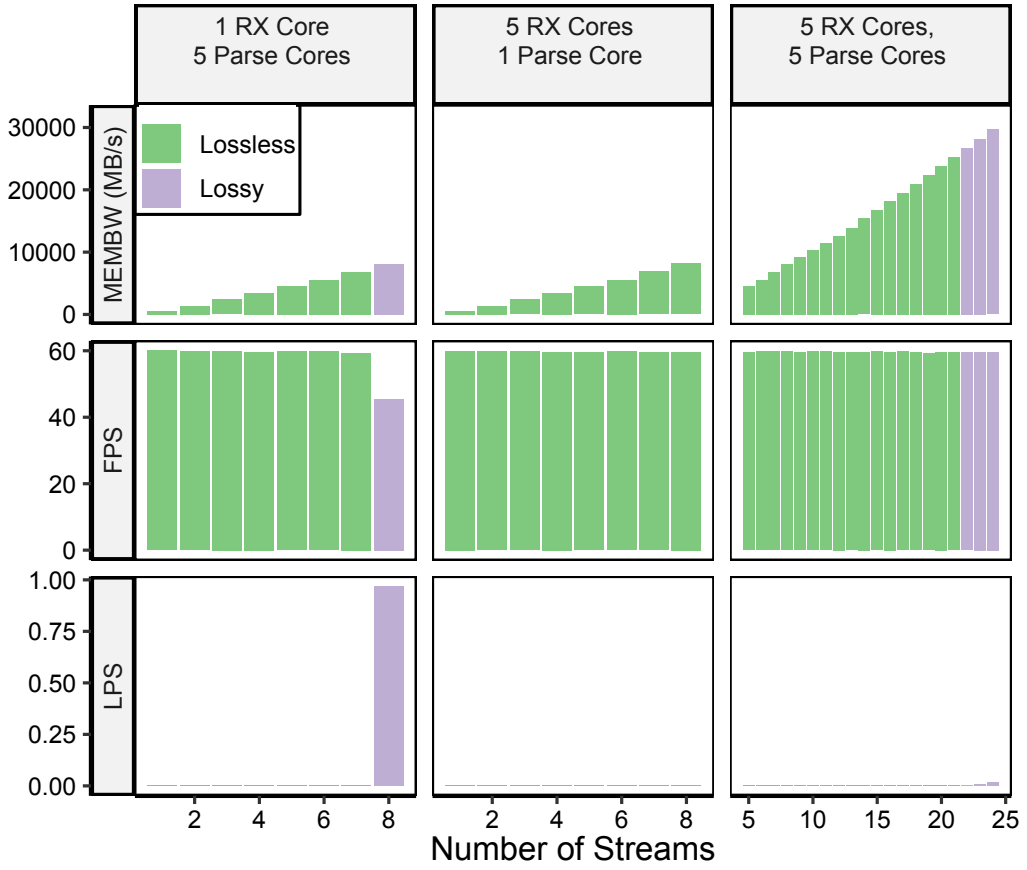


Figure 5.6: vMI performance using VPP, with an application only receiving vMI frames. The optimal allocation is represented in the last column, as the total number of available cores is 12 = 2 VPP Cores + 5 RX Cores + 5 Parse Cores. Because they represent Losses Per Second (LPS), the graphs of the third row show no data points in lossless configurations.

Results

Figures 5.5 and 5.6 summarise the results, with the kernel network-stack and VPP, respectively. In both cases, the optimal CPU core allocation is depicted in the last column, and, the marginal performance of each task is quantified by limiting the corresponding task group to one CPU core.

When using the kernel network-stack, and configuring the NIC with a single queue and a single IRQ core, 3 streams can be received before experiencing packet losses. If the NIC is configured with multiple queues, each

[3]VPP requires a main core (handling all tasks unrelated to packet processing) and a worker core (actually performing packet processing).

receiving a single stream, a single IRQ core is able to process 6 streams. Therefore, all the experiments concerning the kernel network-stack were conducted by ensuring that each stream is received on a different queue, independently from the number of IRQ cores.

Figures 5.5 and 5.6 show that kernel-bypass networking with VPP improves scalability by 40%, as the maximum number of received flows is increased from 15 to 21. Despite zero-copy packet transmission, the VPP-based architecture relies on busy-polling, which introduces a high fixed memory-bandwidth cost. The marginal benefits of zero-copy only appear when receiving more than 4 streams.

Using VPP, between 21 and 24 streams, the observed losses are sporadic, unlike losses appearing above 15 streams when using the kernel network-stack. The former case is explainable by unpredictable latencies leading to queue overflows, whereas the latter case results from low overall packet-processing efficiency.

Finally, when using kernel-bypass, context switches, and hence Last Level Cache (LLC) thrashing, is avoided, improving the performance of the parsing code. That is experimentally supported in figures 5.5 and 5.6, which show that, when using the kernel network-stack, a single parse core can process up to 5 streams before the FPS drops, whereas when using VPP, the FPS remains stable even with 8 received streams.

5.5.3 Full media-processing pipeline

In this section, the impact of the conjunction of kernel-bypass and zero-copy vMI frame transport is quantified on a realistic multi-stage media processing pipeline.

Description

As depicted in figure 5.4, a media-processing pipeline of three stages, each implemented as a vMI-based application, is used.

The *ingress* application includes an SMPTE 2022-6 input and a shared memory output. It is responsible for receiving a media stream from the network and producing a corresponding stream of vMI frames. Those embed video data, encoded as a sequence of 10-bits samples, packed in a contiguous buffer. Therefore, the first video byte contains the eight Most Significant Bits (MSB) of the first sample, the second video byte contains the two Least Significant Bits (LSB) of the first sample, and the six MSB of the second sample, etc. To further ease processing, the *converter* application drops the two LSBs of each sample, to produce vMI frames where each byte is a sample.

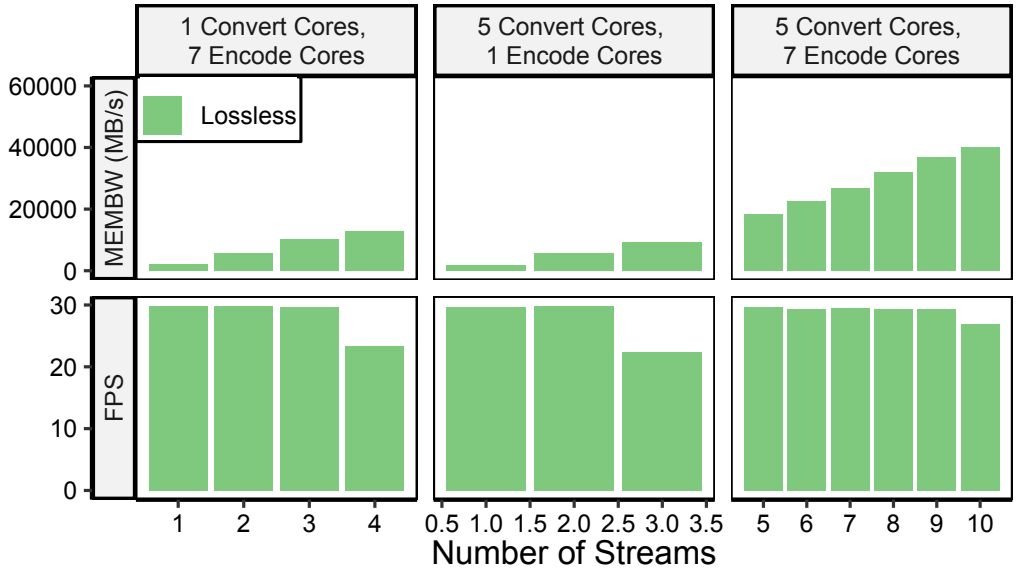


Figure 5.7: vMI performance using the kernel network stack, with a full media processing pipeline. A copy is performed at each stage of the pipeline. The optimal allocation is represented in the last column, as the total number of available cores is 24 = 3 IRQ Cores + 5 RX Cores + 4 Parse Cores + 5 Convert Cores + 7 Encode Cores.

This processing is CPU-and-memory-intensive, as it requires multiple bit-shifting and bit-masking operations, making the converter a realistic facsimile of video processing applications. Finally, an *encoder* application uses the x264 library to transform vMI frames into a compressed H264 video stream, ready for media distribution.

This media-processing pipeline is evaluated in two configurations. First, the SMPTE 2022-6 inputs are using the kernel network-stack and each stage communicates with the next stage with a different shared memory segment, implying vMI-frames copies as stated in section 5.3.1. The second configuration uses VPP-based SMPTE 2022-6 inputs and zero-copy shared memory I/Os. Thus, the first configuration serves as a baseline to quantify the vMI frame transport optimisations introduced in section 5.3.

Additional task groups

The ingress stage of the pipeline requires the same tasks groups as mentioned in section 5.5.2 for microbenchmarking. Those are allocated the same CPU cores as the optimal allocation iteratively determined in section 5.5.2.

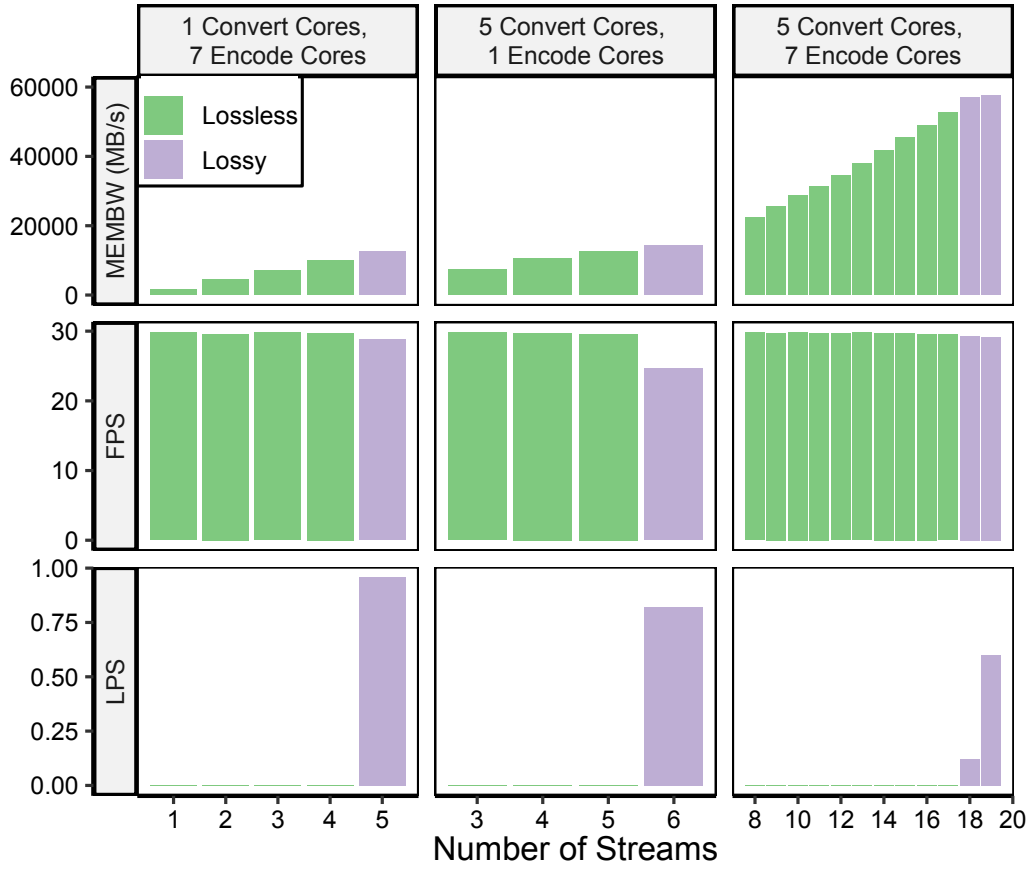


Figure 5.8: vMI performance using VPP and zero-copy, with a full media processing pipeline. The optimal allocation is represented in the last column, as the total number of available cores is 24 = 2 VPP Cores + 5 RX Cores + 5 Parse Cores + 5 Convert Cores + 7 Encode Cores.

Therefore, all the cores of the first CPU are dedicated to reception and parsing of SMPTE 2022-6 streams. The task groups corresponding to 10-bits-to-8-bits video conversion and H264 video encoding are allocated CPU cores from the second CPU, which, in the following, are called *convert cores* and *encode cores*, respectively.

Results

Results are depicted in figure 5.7 for the baseline configuration and in figure 5.8 for the VPP-based zero-copy configuration. In the baseline configuration, as vMI frames are copied between stages, the associated allocated memory can be immediately reused once they reach the vMI output, hence

no apparent losses, even when the convert-and-encode tasks consume excessive resources. In this situation, scalability must be quantified by way of the FPS metric.

In the zero-copy configuration, the lifetime of vMI frames is that of their progress through the whole pipeline (hosted on a single server). As vMI uses a pooled allocator for frames, whenever processing in any stage of the pipeline takes excessive time, packet losses occur as the ingress can allocate no further vMI frames. In this situation, scalability is quantifiable by the FPS and the LPS.

Determined using the method in section 5.5.1, the optimal allocation consists of five convert CPU cores, and seven encode CPU cores, for both configurations. By allocating a single CPU core to the convert task group, the marginal scalability of 10-to-8-bits video conversion is quantified. Figures 5.7 and 5.8 show that VPP and zero-copy operation enable processing 4 streams, whereas the baseline only allowed 3 streams.

Similarly, when using a single encode core, the baseline configuration only allows 2 streams, vs 5 for the one with optimised vMI-frame transport. When considering the performance of the full system, the optimisations presented in section 5.3 increase scalability by 89%, with 17 processed streams vs 9 with the baseline configuration.

Finally, comparing figures 5.8 and 5.7 shows that the proposed optimisations systematically incur a smaller memory-bandwidth footprint, despite the polling overhead of VPP (unlike microbenchmarks). This is explained by the pipeline's depth, which, due to memory-copies, strongly penalises the baseline case, even when considering a single processed stream.

5.6 Conclusion

Until standardisation of SMPTE 2022-6 and 2110, real-time uncompressed media streams for professional broadcast were transported over SDI, with as consequence a limited flexibility due to stressed hardware-dependencies. Even after SMPTE 2022-6 proposed a way of encapsulating SDI data over IP, it has mostly been used to replace video switching with IP routing, as described in chapter 1. Using commodity NICs for processing is challenging, due to the high data-rate and packet-rates involved.

The vMI architecture, proposed in this chapter, was designed, and has been shown, to enable implementation of media processing applications using commodity hardware, without breaking the concepts introduced by APIs coming from the SDI-world. Specifically, a full software-oriented architecture including kernel bypass networking and zero-copy frame transmission

has been specified and implemented, while targeting the use of as generic hardware as possible. The experimental evaluation showed both the feasibility and scalability of vMI. Quantitatively, the conjunction of kernel bypass networking and zero-copy frame transmission improved scalability by 89%, for a realistic media-processing pipeline, hosted on a single server.

From a different perspective, vMI is a key enabler for **intra-server** media transport, either between the NIC and the application, or between two applications. This is in addition to packet-based media transport technologies such as SMPTE 2022-6 and SMPTE 2110, which enable **inter-server** media transport, and which have been shown to be implementable on commodity hardware in chapter 4. Therefore, the joint use of vMI and of packet-based media transport offers a complete solution for media production on commodity servers.

Chapter 6

Conclusion

The increasing performance and parallelism offered by general-purpose CPUs and GPU on commodity servers, as well as the increasing data-rates supported by general-purpose networking equipment — such as Ethernet packet switches — motivated the transition of certain performance-demanding applications **from dedicated hardware to software executed on commodity servers**. Examples of such applications include scientific computing (which was migrated from dedicated super-computers, to networks of commodity servers [147, 148]), and network packet processing (which has been increasingly implemented on commodity servers running Virtual Network Functions (VNF) [149]). Such a migration enables the reuse of the same hardware infrastructure for multiple applications, enabling resource pooling and, ultimately, reducing operational costs.

Prior to the work developed in this thesis, the availability of 10 Gbit/s Ethernet had initiated that transition in the field of professional broadcast and, specifically, in **media production**, *i.e.,* the process of elaborating, in real-time, the audiovisual content that will be consumed by end viewers. That transition had started with the standardisation of network-based — instead of SDI-based — media transport, described in SMPTE 2022-6 and, more completely, in SMPTE 2110. The latter **specified** timing constraints, with which media production packet streams are required to comply. However, as a consequence of those timing constraints, actually **realising** media production with commodity hardware was still an open issue, at it raises multiple challenges.

First, it was generally believed by the broadcasting community, that software-based packet processing introduced a high jitter, making it unsuitable for media production. Prior to this thesis, the validity of that claim was not explicitly analysed and quantified. Furthermore, reducing that jitter was not believed to be technically possible without relying on specialised hard-

ware, capable of packet pacing. Finally, the transition from SDI-based to packet-based media processing was likely to incur large engineering costs, as video-frame processing and packet packet processing are fundamentally different tasks. It was therefore necessary to propose a software framework to ease the implementation of media processing applications, without sacrificing the required performance and scalability.

By addressing all aforementioned challenges, this thesis has, therefore, enabled the migration of media production from dedicated hardware to software executed on commodity servers.

In chapter 2, the jitter introduced by commodity servers upon processing packets belonging to a media production stream has been established to be incompatible with a SMPTE 2022-6 receiver with a limited buffering capacity — which, therefore, rigorously verifies what was, until then, only an informal claim in the broadcasting community. Specifically, on an x86_64 server running a Linux operating system, a minimal jitter in the order of dozens of microseconds was observed, which is too high for some hardware-based receivers (*e.g.,* some IP-to-SDI gateway devices) whose buffering capacity is no higher than 4 packets. Furthermore, that chapter identified all root causes of jitter and classified them into three categories: (i) hardware-related, (ii) network-stack related, and (iii) network-stack independent. Among all the causes, only one has been identified to be impossible to mitigate: the occurrence of System Management Interrupts (SMI). By stealing CPU-cycles (unbeknownst to the operating system's software), SMIs introduce sporadic latency spikes, and thus, set a lower bound on the jitter achievable by commodity servers. Conversely, the relevant steps to disable each other identified jitter source have been enumerated and thus, can easily be exploited to operate media production setups.

More generally, SMIs challenge the accuracy of packet timestamping performed by any kind of software method: an SMI may occur between packet reception and timestamp acquisition and thus, can add an unknown measurement error to the acquired timestamp. In chapter 3, OP4T has been developed as an FPGA-based framework, simplifying the implementation of packet timestamping solutions which, because they are hardware-based, are not affected by the previously-identified jitter sources. Because it is programmable in the P4 language (which is specifically tailored for network packet processing) OP4T can be used by network or broadcasting operators — who are not necessarily specialised in hardware design — to implement a custom packet processing and timestamping logic, which shall be executed on an FPGA-based NIC, with no performance penalty. In particular, OP4T enables packet timestamping with sufficient precision to qualify the compliance of a SMPTE 2110 stream to a certain profile, or to assess that a SMPTE

2022-6 stream has a jitter compatible with a certain receiver buffer size. Furthermore, OP4T was experimentally verified to allow measurements with a microsecond-scale precision when implemented on the NetFPGA-SUME platform, in a simple scenario consisting of evaluating the jitter introduced by a software packet forwarder.

In chapter 4, a solution has been brought to the problem of packet pacing, *i.e.,* the transmission of a packet stream, at a constant-rate and with negligible (microsecond-scale) jitter. Notably, the proposed solution does not require any specific hardware development, and can be built using a commodity server and general-purpose networking equipment. Consequently, it essentially enables software-based transmission of SMPTE 2022-6 and SMPTE 2110 media streams, even to small-buffered receivers with low jitter tolerance. To achieve that goal and circumvent the previously-identified irremediable jitter sources — *i.e.,* the occurrence of SMIs — the concept of assisted-pacing has been introduced, and has been shown to be implementable on commodity hardware, by the clever use of gap-packets, *i.e.,* packets transmitted by a network interface card at a known line-rate, but dropped by the network infrastructure. Furthermore, to ensure that the pacing frequency is exactly the one expected by the receiver, pacing algorithms have been specified, formally proven, and experimentally evaluated. Such an analysis requires an adequate mathematical definition of jitter, which has, therefore, also been proposed in that chapter.

Finally, in chapter 5, the virtual Media Interface (vMI) software architecture has been introduced, as a novel paradigm easing the implementation of high-performance scalable media-processing applications for professional broadcast. Specifically, while a packet-processing application receives and transmits network packets — by relying, *e.g.,* on a socket-like API — a vMI-based media-processing application receives and transmits media frames. Media frames generalises SDI video frames, and correspond to atomic elements, the sequence of which constitutes a media stream. For example, a media frame can be a video frame, an audio sample, or the closed-captions associated with a video frame. A media-processing application has, therefore, to process media frames only and thus, media transport — *e.g.,* over SMPTE 2022-6 or 2110 packet streams — is confined into vMI, and is separated from media-processing. That separation has enabled to transparently bring the benefits of modern packet-processing techniques such as kernel bypass and zero-copy networking to media-processing applications by implementing those techniques once, as part of vMI. Those have been shown to improve the scalability of a typical commodity server by almost doubling the number of streams it can process, in a realistic media encoding scenario.

In addition to academic publications (chapters 2 and 4), and submissions

(chapters 3 and 5), those contributions have led to the release of the work from chapters 4 and 5 as part of the open-source project Cisco Herisson[1]. OP4T is also planned to be released to the community. Furthermore, vMI has been used as part of industrial collaborations with the broadcasting industry, and has been proved to ease the porting of existing SDI-based media-processing appliances, to software running on commodity servers [150, 151].

In summary, the work realised in this thesis has addressed the main challenges occurring when considering the transition of media production from dedicated to commodity hardware. Those challenges range from the scale of packet transmissions (with the understanding, measurement, modelling, and mitigation of network jitter) to the scale of media streams and media-processing applications (with the design of the vMI software architecture). The nature of the vMI software framework has allowed the implementation of media-processing applications as disaggregated pipelines of independent programs, each performing an elementary media-processing operation, and which are distributed across multiple CPU cores, or even, across multiple servers in a data-centre.

As a consequence, the next natural research questions arise from the study of software-based media production at a scale even coarser than media-processing applications, *i.e.,* the study of entire media-production pipelines. Open issues at that scale essentially revolve around the assignment of resources of a media production data centre, to tasks belonging to a set of media production pipelines, as well as around the scheduling of those tasks.

[1] Available at `https://github.com/cisco/herisson`

Appendix A

Mathematical Proofs for Chapter 4

Proof of property 2.

t is (b, f)-paced $\implies$ $J_{ALT}(t) \leq \frac{b}{2f}$

By definition of t being (b, f)-paced, there exist i_0 and u so that, for all $i > i_0$, if $t_i \geq u$:

$$0 \leq (i - i_0) - f \times (t_i - u) \leq b$$

That inequality is invariant by increasing i_0 by an arbitrary integer m and increasing u by $\frac{m}{f}$. Consequently, u and i_0 can be chosen, without loss of generality, so that $u \geq \frac{b}{2f}$. Defining $u' = u - \frac{b}{2f} \geq 0$ yields:

$$-\frac{b}{2} \leq (i - i_0) - f \times (t_i - u') \leq \frac{b}{2}$$

for all $i \geq i_0$ so that $t_i \geq u'$. Hence, by definition of the ALT jitter, $J_{ALT}(t) \leq \frac{b}{2f}$. □

$J_{ALT}(t) < \frac{b}{2f} \implies$ t is (b, f)-paced

By definition of the ALT jitter, there exist i_0 and u so that

$$\sup_{\substack{i \geq i_0 \\ t_i \geq u}} \left| \frac{i - i_0}{f} - (t_i - u) \right| \leq \frac{b}{2f}$$

Defining $u' = u + \frac{b}{2f}$, then for all $i \geq i_0$

$$\begin{aligned}(i - i_0) - f \times (t_i - u') &= (i - i_0) - f \times (t_i - u) + \frac{b}{2} \\ &\leq f \sup_{\substack{i \geq i_0 \\ t_i \geq u}} \left| \frac{i - i_0}{f} - (t_i - u) \right| + \frac{b}{2} \leq b\end{aligned}$$

Also

$$\begin{aligned}(i - i_0) - f \times (t_i - u') &= (i - i_0) - f \times (t_i - u) + \frac{b}{2} \\ &\geq -f \sup_{\substack{i \geq i_0 \\ t_i \geq u}} \left| \frac{i - i_0}{f} - (t_i - u) \right| + \frac{b}{2} \geq 0\end{aligned}$$

Consequently, t is (b, f)-paced. □

$\boldsymbol{\exists i_0, \forall i \geq i_0 : \left| \frac{1}{f} - (t_{i+1} - t_i) \right| \leq \frac{b}{f} \implies J_p(t) \leq \frac{b}{f}}$

By existence of i_0,

$$\sup_{i \geq i_0} \left| \frac{1}{f} - (t_{i+1} - t_i) \right| \leq \frac{b}{f}$$

The peak period jitter $J_p(t)$ is the infimum of that value over all possible i_0, yielding $J_p(t) \leq \frac{b}{f}$. □

$\boldsymbol{J_p(t) < \frac{b}{f} \implies \exists i_0, \forall i \geq i_0 : \left| \frac{1}{f} - (t_{i+1} - t_i) \right| < \frac{b}{f}}$

If $J_p(t) < \frac{b}{f}$, then by definition, there exists i_0 so that for all $i \geq i_0$:

$$\left| \frac{1}{f} - (t_{i+1} - t_i) \right| \leq \frac{1}{2} \left(J_p(t) + \frac{b}{f} \right) < \frac{b}{f}$$

□

□

Proof of property 3.

$\forall j \geq 1 : s_j \in [0, 1) \cup [n_{min}, +\infty)$

The predicate $P(j)$ is defined as $s_j \in [0, 1) \cup [n_{min}, +\infty)$. A proof by induction of $\forall j \geq 1 : P(j)$ is provided in the following. $P(j)$ is verified for $j = 1$, *i.e.,* $P(1)$ is true ($s_1 = 1$). The evolution of s is given by the rules:

$$s_{j+1} = \begin{cases} s_j + \tau_{j+1} - \texttt{dur(p)}, & s_j < 1 \\ s_j - n_{max}, & s_j \geq n_{max} + n_{min} \\ s_j - \lfloor s_j \rfloor, & 1 \leq s_j \leq n_{max} \\ s_j - n_{min}, & \text{otherwise} \end{cases}$$

Reasoning by induction, $P(j)$ is assumed to be true for a given $j > 0$. From the evolution rules, in the first case ($s_j < 1$), $s_j \geq 0$ because of $P(j)$, and combining with precondition $\tau_{j+1} - \texttt{dur(p)} \geq n_{min}$ yields $s_{j+1} \geq n_{min}$, hence $P(j+1)$. In the second and third case, $P(j+1)$ is trivially true. In the fourth case, $s_j \in]n_{max}, n_{max} + n_{min}[$ because the evolution would have fallen into one of the previous cases otherwise. Hence, $s_{j+1} = s_j - n_{min} \in]n_{max} - n_{min}, n_{max}[$. As per condition (4.3), $n_{max} - n_{min} > n_{min}$, which proves $P(j + 1)$. Thus, by induction, $P(j)$ is true for all $j > 0$. □

$s_j \geq 1 \implies n_{min} \leq s_j - s_{j+1} \leq n_{max}$

If $s_j \geq 1$, the case hit by the evolution rule is the second, the third, or the fourth one. In the second case, $s_j - s_{j+1} = n_{max}$. In the third case, $s_j - s_{j+1} = \lfloor s_j \rfloor$, and $\lfloor s_j \rfloor \geq n_{min}$ per $P(j)$ and $\lfloor s_j \rfloor \leq n_{max}$ by definition of the third case. In the fourth case, $s_j - s_{j+1} = n_{min}$. □

□

Proof of property 4. s_j and τ_j are defined as the values of state variables s and τ_{cur} at the beginning the j-th iteration of the main loop. N_j is defined as the total count of enqueued `send` operations. By an analysis similar to the one performed in section 4.4.2:

$$s_j = \sum_{i=1}^{N_j} \tau_{j_i} - y_j \tag{A.1}$$

Algorithm 2 being derived from algorithm 1 with varying τ, it is natural to analyse the evolution of τ_{cur}. τ_{cur} is only updated at an iteration of the loop corresponding to $s < 1$, *i.e.,* a packet transmission. Consequently, it is possible to define i_k so that the k-th update of τ_{cur} happens at the same iteration as the one sending the i_k-th packet. Thus between the j_{i_k}-th and

$(j_{i_{k+1}} - 1)$-th iteration of the loop, the value of τ_{cur} is constant. Considering equation (A.1) for $i \in [i_k, i_{k+1} - 1]$ and noting that $N_{j_i} = i - 1$:

$$s_{j_i} - s_{j_{i_k}} = (i - i_k)\tau_{j_{i_k}} - (y_{j_i} - y_{j_{i_k}}) \tag{A.2}$$

From algorithm 2, line 7, and as $s_{j_i} \in [0, 1[$

$$\begin{cases} y_{j_i} - y_{j_{i_k}} = \left\lfloor s_{j_{i_k}} + (i - i_k)\frac{y_{j_{i_k}} - y_{j_{i_{k-1}}}}{F(W+y_{j_{i_k}}) - F(W+y_{j_{i_{k-1}}})} \right\rfloor \\ i_{k+1} = \min\left\{ l > i_k \;\middle|\; y_{j_l} - y_{j_{i_k}} \geq W \right\} \end{cases} \tag{A.3}$$

As $s_{j_1} = 0$ and $y_{j_{i_1}} - y_{j_{i_0}} = W$, it follows from A.3 by a trivial induction that, for all $k \geq 1$, $s_{j_{i_k}} = 0$, $y_{j_{i_k}} - y_{j_{i_{k-1}}} = W$, and $N_{j_{i_{k+1}}} - N_{j_{i_k}} = F(W + y_{j_{i_k}}) - F(W + y_{j_{i_{k-1}}})$. Thus, summing over k yields, for all $k \geq 2$, $y_{j_{i_k}} = W(k-1)$, $N_{j_{i_k}} = F(W(k-1))$. Finally, from the first equation of (A.3):

$$y_{j_i} = \left\lfloor W \frac{i - 1 - F(W(k(i) - 1))}{F(Wk(i)) - F(W(k(i) - 1))} \right\rfloor + W(k(i) - 1) \tag{4.8}$$

□

Proof of property 5. y_{j_i} and $k(i)$ are defined as in section 4.4.3. w_i is defined as $w_i = W(k(i) - 1)$. For all $i_0 \in \mathbb{N}$,$y_0 > 0$, and $i > i_0$ so that $y_{j_i} \geq y_0$, the following holds:

$$\begin{aligned} &\tau(i - i_0) - (y_{j_i} - y_0) && \text{(A.4)} \\ &= \tau(i - i_0) - \left(\left\lfloor W \frac{i - 1 - F(w_i)}{F(W + w_i) - F(w_i)} \right\rfloor + w_i - y_0 \right) \\ &= \tau i - \left(W \frac{i - 1 - F(w_i)}{F(W + w_i) - F(w_i)} + w_i - M_i + \tau i_0 - y_0 \right) \\ &= \left(\left(\tau - \frac{W}{F(W + w_i) - F(w_i)} \right) (i - 1 - F(w_i)) \right) \\ &\qquad + (\tau(F(w_i) + 1) - w_i) + M_i - \tau i_0 + y_0 && \text{(A.5)} \end{aligned}$$

with M_i, some fractional part bounded by 1. Consider $\epsilon > 0$. By definition of $A_W(F)$ and $C_W(F)$, there exists $k_1 \in \mathbb{N}$ and $k_0 \in \mathbb{N}$ so that:

$$\sup_{k > k_0} |\tau(F(W(k+1)) - F(Wk)) - W| \leq \frac{A_W(F)}{T^e} + \epsilon$$

$$\sup_{k > k_0} |\tau F(kW) - kW + k_1 W| \leq \frac{C_W(F)}{T^e} + \epsilon$$

Choosing $i_0 = k_0$, $y_0 = \tau i_0 + k_1 W$, and applying to equation (A.5) yields:

$$\begin{aligned} &\sup_{\substack{i>i_0 \\ y_{j_i}>y_0}} |\tau(i - i_0) - (y_{j_i} - y_0)| \\ &\leq \frac{A_W(F)}{T^e} + \frac{C_W(F)}{T^e} + 1 + 2\epsilon \end{aligned}$$

As such i_0 and y_0 may be constructed for all $\epsilon > 0$, the infimum of that last equation over all possible y_0 and i_0, and multiplying by T^e gives:

$$J_{ALT}(t^p) \leq A_W(F) + C_W(F) + T^e$$

□

Appendix B

Résumé en français

La production de média pour la diffusion audiovisuelle (i.e., le processus par lequel plusieurs sources audiovisuelles sont mélangées et traitées en temps réel pour élaborer le flux consommé par le téléspectateur) est généralement implémentée par du matériel dédié, basé sur la Serial Digital Interface (SDI), une technologie d'interconnection dérivé de la télévision analogique. Malgré l'effort industriel présent pour remplacer le SDI par de l'IP (ainsi que spécifié par les standards SMPTE 2022-6 et 2110) la sensibilité au délai de la production de média rend difficile une transition totale vers un traitement logiciel sur des serveurs générique. Cette thèse résout different aspects de ce problème.

Premièrement, il a été conduit une étude quantitative et qualitative de la gigue subie par ces flux lors d'un traitement logiciel. Au delà de résultats obtenus pour des serveurs Linux x86_64, il a été dérivé une méthodologie générale, applicable à tout système d'exploitation et architecture matérielle, permettant d'étudier la gigue introduite. En particulier, il a été montré que la gigue induite par les serveurs Linux x86_64 était de l'ordre de plusieurs dizaines de microsecondes, et ce indépendamment de toute optimisation logicielle implémentée au niveau du système d'exploitation. Ceci est due à un certain type d'interruption, les System Management Interrupts (SMIs) qui entrainent régulièrement l'exécution de code spécifique à la platforme matérielle, et ce indépendamment de la volonté du système d'exploitation (execution dite en System Management Mode, ou ring -1)

Deuxièmement, une plateforme générique a été proposé afin de permettre la réalisation de système d'instrumentation personnalisé, pour l'horodatage précis de packet réseaux. Bien qu'étant basée sur la technologie des FPGA, cette plateforme permet à tout opérateur réseau ou de diffusion audiovisuelle de spécifier une logique d'horodatage personnalisée en utilisant le langage P4. Cela permet en particulier la conception d'une instrumentation pour

la qualification de flux média. Cette plateforme a été mise à l'épreuve lors d'une étude de cas, consistant à évaluer la latence et la gigue introduite par un système de traitement de paquets purement logiciel, disponible en open-source : le Vector Packet Processor (VPP). Plus particulièrement, une logique d'horodatage adaptée a été specifiée en P4, et le système ainsi obtenu a été expérimentalement évalué. Les résultats obtenus montrent que la plateforme proposée permet une precision optimale (de l'ordre de un cycle de FPGA), et qu'elle permet d'observer le comportement de VPP avec une granularité très fine. En particulier, le traitement de paquet par lots, caractéristique très spécifique à VPP) a pu être retrouvé par une simple observation de les séries temporelles obtenues par le système d'horodatage.

Troisièmement, un système de lissage de traffic (packet-pacing) a été proposé, afin de permettre l'envoi de flux de paquets avec une gigue négligeable. Malgré un emploi exclusif de matériel générique, il a été prouvé formellement et expérimentation que la gigue ainsi obtenue était suffisamment faible pour des flux média. Spécifiquement, le système de lissage surmonte les limitations inhérentes aux serveurs génériques (comme les SMIs mentionnés ci-dessus) en adoptant une approche de lissage assisté, c'est à dire, l'emploi d'un composant externe (Pacing-Assistant) qui réalisera les operations d'envoi de paquet et d'attente entre chaque paquet, sans être contraint par les SMIs. Il a été montré que ce composant externe pouvait effectivement être réalisé avec du matériel générique, par exemple la conjonction d'une interface réseau physique et d'un commutateur de paquets standard.

Finalement, un cadre logiciel facilitant l'écriture d'applications de traitement média a été proposé. Ce cadre repose sur la séparation entre le traitement et le transport des flux média, la couche de transport s'occupant du traitement haute performance des paquets réseaux par l'emploi de techniques comme le zero-copy, ou le kernel-bypass. Ces techniques ont été expérimentalement évaluées, et leurs avantages ont été prouvés dans un contexte de passage à l'échelle.

List of Figures

List of Tables

LIST OF TABLES

List of Algorithms

LIST OF ALGORITHMS

Bibliography

[1] G. F. Pfister, "An introduction to the infiniband architecture."

[2] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Enabling Scalable High-Performance Systems with the Intel Omni-Path Architecture," vol. 36, no. 4, pp. 38–47.

[3] InfiniBand Trade Association, "InfiniBand Architecture specification, volume 1, release 1.0."

[4] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson, "Cost-benefit analysis of Cloud Computing versus desktop grids," in *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–12.

[5] C. Vecchiola, S. Pandey, and R. Buyya, "High-Performance Cloud Computing: A View of Scientific Applications," in *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pp. 4–16.

[6] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," vol. 22, no. 6, pp. 931–945.

[7] S. Hazelhurst, "Scientific computing using virtual high-performance computing: A case study using the Amazon elastic computing cloud," in *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, ser. SAICSIT '08. Association for Computing Machinery, pp. 94–103. [Online]. Available: https://doi.org/10.1145/1456659.1456671

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," vol. 5, no. 8.

[9] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." vol. 8, p. 121.

[10] J. Maisonneuve, M. Deschanel, J. Heiles, W. Li, H. Liu, R. Sharpe, and Y. Wu, "An Overview of IPTV Standards Development," vol. 55, no. 2, pp. 315–328.

[11] S. Wright, S. Jones, and C. S. Lee, "IPTV systems, standards, and architectures: Part I [Guest Editorial]," vol. 46, no. 2, pp. 69–69.

[12] ——, "Guest Editorial - IPTV Systems, Standards and Architectures: Part II," vol. 46, no. 5, pp. 92–93.

[13] H. J. Kim and S. G. Choi, "A study on a QoS/QoE correlation model for QoE evaluation on IPTV service," in *2010 The 12th International Conference on Advanced Communication Technology (ICACT)*, vol. 2, pp. 1377–1382.

[14] H. Nam, K.-H. Kim, J. Y. Kim, and H. Schulzrinne, "Towards QoE-aware video streaming using SDN," in *2014 IEEE Global Communications Conference*, pp. 1317–1322.

[15] I. Sodagar, "The MPEG-DASH Standard for Multimedia Streaming Over the Internet," vol. 18, no. 4, pp. 62–67.

[16] R. Pantos and W. May. HTTP Live Streaming. [Online]. Available: https://tools.ietf.org/html/rfc8216

[17] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An analysis of Internet content delivery systems," vol. 36, pp. 315–327. [Online]. Available: https://doi.org/10.1145/844128.844158

[18] S. Gadde, J. Chase, and M. Rabinovich, "Web caching and content distribution: A view from the interior," vol. 24, no. 2, pp. 222–231. [Online]. Available: https://doi.org/10.1016/S0140-3664(00)00318-2

[19] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek, "The measured performance of content distribution networks," vol. 24, no. 2, pp. 202–206. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366400003157

[20] C. Cranor, M. Green, C. Kalmanek, D. Shur, S. Sibal, J. Van der Merwe, and C. Sreenan, "Enhanced streaming services in a content distribution network," vol. 5, no. 4, pp. 66–75.

[21] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky, "A transport layer for live streaming in a content delivery network," vol. 92, no. 9, pp. 1408–1419.

[22] T. C. Thang, Q.-D. Ho, J. W. Kang, and A. T. Pham, "Adaptive streaming of audiovisual content using MPEG DASH," vol. 58, no. 1, pp. 78–85.

[23] J. Kua, G. Armitage, and P. Branch, "A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP," vol. 19, no. 3, pp. 1842–1866, thirdquarter 2017.

[24] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoß feld, and P. Tran-Gia, "A Survey on Quality of Experience of HTTP Adaptive Streaming," vol. 17, no. 1, pp. 469–492, Firstquarter 2015.

[25] M. Dąbrowski, R. Koł odyński, and W. Zieliński, "Analysis of video delay in Internet TV service over adaptive HTTP streaming," pp. 143–150. [Online]. Available: https://fedcsis.org/proceedings/2015/drp/91.html

[26] "ST 292-1:2012 - SMPTE Standard - 1.5 Gb/s Signal/Data Serial Interface," pp. 1–20.

[27] "ST 274:2008 - SMPTE Standard - Television — 1920 x 1080 Image Sample Structure, Digital Representation and Digital Timing Reference Sequences for Multiple Picture Rates," pp. 1–35.

[28] "ST 259:2008 - SMPTE Standard - For Television — SDTV - Digital Signal/Data — Serial Digital Interface," pp. 1–18.

[29] "ST 125:2013 - SMPTE Standard - SDTV Component Video Signal Coding 4:4:4 and 4:2:2 for 13.5 MHz and 18 MHz Systems," pp. 1–26.

[30] W. Feingold, "Color Television - A Primer on the NTSC System," vol. PGBTR-4, no. 1, pp. 30–37.

[31] "ST 240:1999 - SMPTE Standard - For Television — 1125-Line High-Definition Production Systems — Signal Parameters," pp. 1–7.

[32] K. P. Davies, "Some Concepts for the Digital Television Studio," vol. 93, no. 1, pp. 18–23.

[33] M. M. Guess, "Design Considerations for Digital Television Studios: The Transition Phase," vol. 100, no. 12, pp. 955–960.

[34] F. Davidoff, "The All-Digital Television Studio," vol. 89, no. 6, pp. 445–449.

[35] B. E. Phelps, "Magnetic recording method," patentus 2 774 646A. [Online]. Available: https://patents.google.com/patent/US2774646/en

[36] "ST 291-1:2011 - SMPTE Standard - Ancillary Data Packet and Space Formatting," pp. 1–17.

[37] "ST 272:2004 - SMPTE Standard - For Television — Formatting AES Audio and Auxiliary Data into Digital Video Ancillary Data Space," pp. 1–17.

[38] "ST 299-1:2009 - SMPTE Standard - 24-Bit Digital Audio Format for SMPTE 292 Bit-Serial Interface," pp. 1–28.

[39] J. Roizen, "Teletext in the USA," vol. 90, no. 7, pp. 602–610.

[40] "ST 334-1:2015 - SMPTE Standard - Vertical Ancillary Data Mapping of Caption Data and other Related Data," pp. 1–8.

[41] "ST 334-2:2015 - SMPTE Standard - Caption Distribution Packet (CDP) Definition," pp. 1–13.

[42] J. Vallee, M. Artigalas, and M. Favreau, "Digital Production Switchers," vol. 95, no. 3, pp. 295–300.

[43] "IEEE Standard for Ethernet," pp. 1–4017.

[44] "ST 2022-6:2012 - SMPTE Standard - Transport of High Bit Rate Media Signals over IP Networks (HBRMT)," pp. 1–16.

[45] J. Postel. User Datagram Protocol. [Online]. Available: https://tools.ietf.org/html/rfc768

[46] V. Jacobson, R. Frederick, S. Casner, and H. Schulzrinne. RTP: A Transport Protocol for Real-Time Applications. [Online]. Available: https://tools.ietf.org/html/rfc3550

[47] "ST 2059-1:2015 - SMPTE Standard - Generation and Alignment of Interface Signals to the SMPTE Epoch," pp. 1–31.

[48] *ST 2059-2:2015 : SMPTE Profile for Use of IEEE-1588 Precision Time Protocol in Professional Broadcast Applications*. The Society of Motion Picture and Television Engineers.

[49] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," pp. 1–300.

[50] "VSF Technical Recommendation TR-04 - Utilization of ST-2022-6 Media Flows within a VSF TR-03 Environment."

[51] S. Kunić and Z. Šego, "Analysis of Television Technology Transformation from SDI to IP Production," in *2018 International Symposium ELMAR*, pp. 211–216.

[52] "OV 2110-0:2018 - SMPTE Overview Document - Professional Media over Managed IP Networks Roadmap for the 2110 Document Suite," pp. 1–4.

[53] "ST 2110-20:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: Uncompressed Active Video," pp. 1–22.

[54] "ST 2110-30:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: PCM Digital Audio," pp. 1–9.

[55] "ST 2110-40:2018 - SMPTE Standard - Professional Media Over Managed IP Networks: SMPTE ST 291-1 Ancillary Data," pp. 1–8.

[56] "ST 2110-10:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: System Timing and Definitions," pp. 1–17.

[57] "Xilinx SMPTE 2022-5/6 Video over IP Transmitter v4.0." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/v_smpte2022_56_tx/v4_0/pg032_v_smpte2022_56_tx.pdf

[58] "Cisco Nexus 9508 Switch Performance Test." [Online]. Available: https://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-9000-series-switches/cisco-report-nexus-9508.pdf

[59] "ST 2110-21:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: Traffic Shaping and Delivery Timing for Video," pp. 1–17.

[60] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," vol. 44, no. 3, pp. 87–95. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[61] E. Vicente and R. Matias Jr., "Exploratory Study on the Linux OS Jitter," in *2012 Brazilian Symposium on Computing System Engineering*. IEEE, pp. 19–24. [Online]. Available: http://ieeexplore.ieee.org/document/6473626/

[62] 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*, pp. 852–863.

[63] P. De, R. Kothari, and V. Mann, "Identifying sources of Operating System Jitter through fine-grained kernel instrumentation," in *2007 IEEE International Conference on Cluster Computing*. IEEE, pp. 331–340. [Online]. Available: http://ieeexplore.ieee.org/document/4629247/

[64] 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," vol. 55, no. 1, pp. 435–439. [Online]. Available: http://ieeexplore.ieee.org/document/4448543/

[65] 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: MEASURING AND REDUCING THE IMPACT OF THE OS ON END-TO-END LATENCIES," vol. 42, no. 11, pp. 1315–1330. [Online]. Available: http://doi.wiley.com/10.1002/spe.1134

[66] "Intel® Ethernet Controller XL710: Documents and Datasheets." [Online]. Available: https://www.intel.com/content/www/uk/en/design/products-and-solutions/networking-and-io/ethernet-controller-xl710/technical-library.html

[67] J. Woodruff, A. W. Moore, and N. Zilberman, "Measuring Burstiness in Data Center Applications," in *Proceedings of the 2019 Workshop on Buffer Sizing*. ACM, pp. 1–6. [Online]. Available: http://dl.acm.org/doi/10.1145/3375235.3375240

[68] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," vol. 40, no. 4, pp. 63–74. [Online]. Available: https://doi.org/10.1145/1851275.1851192

[69] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. Association for Computing Machinery, pp. 267–280. [Online]. Available: https://doi.org/10.1145/1879141.1879175

[70] D. A. Popescu, N. Zilberman, and A. W. Moore, "Characterizing the impact of network latency on cloud-based applications' performance." [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-914.html

[71] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. Association for Computing Machinery, pp. 78–85. [Online]. Available: https://doi.org/10.1145/3131365.3131375

[72] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09. Association for Computing Machinery, pp. 202–208. [Online]. Available: https://doi.org/10.1145/1644893.1644918

[73] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: Open source network tester," vol. 28, no. 5, pp. 6–12.

[74] A. Oeldemann, T. Wild, and A. Herkersdorf, "FlueNT10G: A Programmable FPGA-Based Network Tester for Multi-10-Gigabit Ethernet," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 178–1787.

[75] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," vol. 34, no. 5, pp. 32–41. [Online]. Available: http://ieeexplore.ieee.org/document/6866035/

[76] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15*. ACM Press, pp. 275–287. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815675.2815692

[77] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. Association for Computing Machinery, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3289602.3293924

[78] K. Vipin and S. A. Fahmy, "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications." [Online]. Available: https://doi.org/10.1145/3193827

[79] P. Banerjee, M. Sangtani, and S. Sur-Kolay, "Floorplanning for Partially Reconfigurable FPGAs," vol. 30, no. 1, pp. 8–17.

[80] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, "Floorplanning Automation for Partial-Reconfigurable FPGAs via Feasible Placements Generation," vol. 25, no. 1, pp. 151–164.

[81] M. Rabozzi, J. Lillis, and M. D. Santambrogio, "Floorplanning for Partially-Reconfigurable FPGA Systems via Mixed-Integer Linear Programming," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 186–193.

[82] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-Speed Software Data Plane via Vectorized Packet Processing," vol. 56, no. 12, pp. 97–103.

[83] R. Benice and A. Frey, "An Analysis of Retransmission Systems," vol. 12, no. 4, pp. 135–145. [Online]. Available: http://ieeexplore.ieee.org/document/1088975/

[84] J. Postel, "Transmission Control Protocol." [Online]. Available: https://www.rfc-editor.org/info/rfc0793

[85] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat.*

No.00CH37064), vol. 3. IEEE, pp. 1157–1165. [Online]. Available: http://ieeexplore.ieee.org/document/832483/

[86] M. Ghobadi and Y. Ganjali, "TCP Pacing in Data Center Networks," in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*. IEEE, pp. 25–32. [Online]. Available: http://ieeexplore.ieee.org/document/6627732/

[87] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The E ects of Two-Way Tra c," in *Proceedings of ACM Sigcomm*, vol. 91, pp. 133–147.

[88] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and Van Jacobson, "BBR: Congestion-based congestion control," vol. 60, no. 2, pp. 58–66. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3042068.3009824

[89] Y. Cai, T. Wolf, and W. Gong, "Delaying Transmissions in Data Communication Networks to Improve Transport-Layer Performance," vol. 29, no. 5, pp. 916–927. [Online]. Available: http://ieeexplore.ieee.org/document/5753556/

[90] V. Sivaraman, H. Elgindy, D. Moreland, and D. Ostry, "Packet Pacing in Small Buffer Optical Packet Switched Networks," vol. 17, no. 4, pp. 1066–1079. [Online]. Available: http://ieeexplore.ieee.org/document/4895292/

[91] P. Thiran, J.-Y. Le Boudec, and F. Worm, "Network calculus applied to optimal multimedia smoothing," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3. IEEE, pp. 1474–1483. [Online]. Available: http://ieeexplore.ieee.org/document/916643/

[92] J. D. Salehi, Z.-L. Zhang, J. F. Kurose, and D. Towsley, "Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing," vol. 24, no. 1, pp. 222–231. [Online]. Available: http://portal.acm.org/citation.cfm?doid=233008.233047

[93] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable Traffic Shaping at End Hosts," in *Proceedings of the Conference of the ACM Special Interest*

Group on Data Communication - SIGCOMM '17. ACM Press, pp. 404–417. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098822.3098852

[94] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa, "Design and evaluation of precise software pacing mechanisms for fast long-distance networks."

[95] R. Takano, T. Kudoh, Y. Kodama, and F. Okazaki, "High-resolution timer-based packet pacing mechanism on the linux operating system," vol. 94, no. 8, pp. 2199–2207.

[96] D. Intel, "Data plane development kit."

[97] N. A. Lynch and M. R. Tuttle, "An introduction to Input/Output automata," vol. 2, no. 3, pp. 219–246.

[98] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," vol. 93, p. 110.

[99] S. Müller, "CPU Time Jitter Based Non-Physical True Random Number Generator," in *Linux Symposium*. Citeseer, p. 23.

[100] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT," in *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. SYSGO AG, pp. 19–29.

[101] J. H. Brown and B. Martin, "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications," in *Proc. of the 12th Real-Time Linux Workshop (RTLWS'12)*, pp. 1–17.

[102] B. Delgado and K. L. Karavanic, "Performance implications of System Management Mode," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, pp. 163–173. [Online]. Available: http://ieeexplore.ieee.org/document/6704682/

[103] A. Toussaint, M. Hawari, and T. Clausen, "Chasing Linux Jitter Sources for Uncompressed Video," in *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 395–401.

[104] J.-L. Ferrant, M. Gilson, S. Jobert, M. Mayer, M. Ouellette, L. Montini, S. Rodrigues, and S. Ruffini, "Synchronous ethernet: A method to transport synchronization," vol. 46, no. 9, pp. 126–134.

[105] G. ITU, "8261-Timing and synchronization aspects in packet networks."

[106] "ANY-FREQUENCY PRECISION CLOCK MULTIPLIER/ JITTER ATTENUATOR." [Online]. Available: https://www.silabs.com/documents/public/data-sheets/Si5324.pdf

[107] B. J. Olsson, "IP QoS Objectives For Broadcast Services," in *NAB 2014 Conference Contribution (May Be Retrieved at https://www. Researchgate. Net/Publication/261476525_IP_QoS_Object Ives_For_-Broadcast_Services).*

[108] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," vol. 20, no. 4, pp. 3645–3676.

[109] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX Association, pp. 9–9.

[110] T. Hø iland Jø rgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. ACM, pp. 54–66. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443

[111] Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration - IEEE Conference Publication. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7387409

[112] I. Cerrato, M. Annarumma, and F. Risso, "Supporting Fine-Grained Network Functions through Intel DPDK," pp. 1–6.

[113] R. Bonafiglia, I. Cerrato, F. Ciaccia, M. Nemirovsky, and F. Risso, "Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking," in *2015 Fourth European Workshop on Software Defined Networks*, pp. 67–72.

[114] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK," in *2013 Second European Workshop on Software Defined Networks*, pp. 62–67.

[115] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore

systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pp. 489–502.

[116] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, pp. 175–186.

[117] "FD.Io VPP: Shared Memory Packet Interface (memif) Library." [Online]. Available: https://docs.fd.io/vpp/19.08/libmemif_doc.html

[118] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz, "Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges," vol. 55, no. 2, pp. 216–223.

[119] The actual cost of software switching for NFV chaining - IEEE Conference Publication. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7987296

[120] Anthony, S. R. Chowdhury, T. Bai, R. Boutaba, and J. François, "UNiS: A User-Space Non-Intrusive Workflow-Aware Virtual Network Function Scheduler," in *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 152–160.

[121] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, "$\mu\mathrmNF$: A Disaggregated Packet Processing Architecture," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, pp. 342–350.

[122] M.-A. Kourtis, H. Koumaras, G. Xilouris, and F. Liberal, "An NFV-Based video quality assessment method over 5G small cell networks," vol. 24, no. 4, pp. 68–78.

[123] B.-M. Andrus, S. A. Sasu, T. Szyrkowiec, A. Autenrieth, M. Chamania, J. K. Fischer, and S. Rasp, "Zero-touch provisioning of distributed video analytics in a software-defined metro-haul network with P4 processing," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE, pp. 1–3.

[124] M. Azimi, R. Boitard, M. T. Pourazad, and P. Nasiopoulos, "Performance evaluation of single layer HDR video transmission pipelines," vol. 63, no. 3, pp. 267–276.

[125] J. Lapierre and M. Al-Habbal, "Bridging the gap between software and SMPTE ST 2110," in *SMPTE 2018*. SMPTE, pp. 1–7.

[126] L. Han, J. Yan, and Y. Cai, "An implementation of capture and playback for IP-Encapsulated video in professional media production," in *International Forum on Digital TV and Wireless Multimedia Communications*. Springer, pp. 346–355.

[127] N. Ranasinghe, R. Bangamuarachchi, J. Seneviratne, A. Jayawardane, A. Pasqual, and R. M. A. U. Senarath, "SMPTE ST 2110 Compliant Scalable Architecture on FPGA for end to end Uncompressed Professional Video Transport Over IP Networks," in *2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, pp. 235–238.

[128] "Linux Media Subsystem." [Online]. Available: https://www.kernel.org/doc/html/latest/media/index.html#linux-media-subsystem-documentation

[129] "Video For Linux API." [Online]. Available: https://www.kernel.org/doc/html/latest/media/uapi/v4l/v4l2.html

[130] "Buffer Sharing and Synchronization." [Online]. Available: https://www.kernel.org/doc/html/v4.16/driver-api/dma-buf.html

[131] "Software Development Kits - Deltacast." [Online]. Available: https://www.deltacast.tv/products/developer-products/software-development-kits

[132] "Matrox Video - Developer Products for PC." [Online]. Available: https://www.matrox.com/video/en/products/developer/software/

[133] "AJA Video Systems." [Online]. Available: https://www.aja.com/family/developer

[134] "AJA NTV2 V4L2/ALSA Driver," AJA Video Systems Inc. [Online]. Available: https://github.com/aja-video/ntv2-v4l2

[135] "Xilinx Zynq-7000 SoC ZC702 Base Targeted Reference Design." [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/2015_4/ug925-zynq-zc702-base-trd.pdf

[136] "Xilinx/Linux-Xlnx," Xilinx. [Online]. Available: https://github.com/Xilinx/linux-xlnx

[137] G. Team, *Gstreamer*.

[138] E. Walthinsen, "GStreamer-GNOME Goes Multimedia."

[139] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond Softnet," in *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ser. ALS '01. USENIX Association, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268488.1268506

[140] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," vol. 15, no. 3, pp. 217–252. [Online]. Available: http://portal.acm.org/citation.cfm?doid=263326.263335

[141] M. Gorman, "Huge pages part 1 (Introduction)."

[142] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. ACM, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[143] H. Subramoni, P. Lai, M. Luo, and D. K. Panda, "RDMA over Ethernet — A preliminary study," in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–9.

[144] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," vol. 72, no. 11, pp. 1471–1480. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512000329

[145] J. D. Valois, "Lock-free Linked Lists Using Compare-and-Swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. ACM, pp. 214–222. [Online]. Available: http://doi.acm.org/10.1145/224964.224988

[146] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ser. ISCA '84. ACM, pp. 348–354. [Online]. Available: http://doi.acm.org/10.1145/800015.808204

[147] M. Baker, G. Fox, and H. Yau, "Cluster Computing Review." [Online]. Available: https://surface.syr.edu/npac/33

[148] M. Baker and R. Buyya, "Cluster computing: The commodity supercomputer," vol. 29, no. 6, pp. 551–576.

[149] Y. Li and M. Chen, "Software-Defined Network Function Virtualization: A Survey," vol. 3, pp. 2542–2553.

[150] Cisco, Intel and Microsoft Accelerate the Transition from SDI to IP with an Open Source Toolkit for Media Software Vendors. Cisco Blogs. [Online]. Available: https://blogs.cisco.com/sp/cisco-intel-and-microsoft-sdi-to-ip

[151] DELTACAST expands IP collaboration with CISCO - Deltacast. [Online]. Available: https://www.deltacast.tv/news/news/2018/deltacast-expands-ip-collaboration-with-cisco

BIBLIOGRAPHY

ECOLE
DOCTORALE

Titre : Aspects réseaux et systèmes de la migration d'application hautes performances de matériel dédié vers des serveurs génériques : l'exemple de la production de média pour la diffusion audiovisuelle

Mots clés : traitement média, lissage de trafic, gigue, instrumentation, serveurs standards, architecture logicielle

Résumé : La production de média pour la diffusion audiovisuelle (i.e., le processus par lequel plusieurs sources audiovisuelles sont mélangées et traitées en temps réel pour élaborer le flux consommé par le téléspectateur) est généralement implémentée par du matériel dédié, basé sur la Serial Digital Interface (SDI), une technologie d'interconnection dérivé de la télévision analogique. Malgré l'effort industriel présent pour remplacer le SDI par de l'IP (ainsi que spécifié par les standards SMPTE 2022-6 et 2110) la sensibilité au délai de la production de média rend difficile une transition totale vers un traitement logiciel sur des serveurs générique. Cette thèse résout different aspects de ce problème. Premièrement, il a été conduit une étude quantitative et qualitative de la gigue subie par ces flux lors d'un traitement logiciel. Au delà de résultats obtenus pour des serveurs Linux x86_64, il a été dérivé une méthodologie générale, applicable à tout système d'exploitation et architecture matérielle, permettant d'étudier la gigue introduite. Deuxièmement, une plateforme générique a été proposé afin de permettre la réalisation de système d'instrumentation personnalisé, pour l'horodatage précis de packet réseaux. Bien qu'étant basée sur la technologie des FPGA, cette plateforme permet à tout opérateur réseau ou de diffusion audiovisuelle de spécifier une logique d'horodatage personnalisée en utilisant le langage P4. Cela permet en particulier la conception d'une instrumentation pour la qualification de flux média. Troisièmement, un système de lissage de traffic (packet-pacing) a été proposé, afin de permettre l'envoi de flux de paquets avec une gigue négligeable. Malgré un emploi exclusif de matériel générique, il a été prouvé formellement et expérimentalement que la gigue ainsi obtenue était suffisamment faible pour des flux média. Finalement, un cadre logiciel facilitant l'écriture d'applications de traitement média a été proposé. Ce cadre repose sur la séparation entre le traitement et le transport des flux média, la couche de transport s'occupant du traitement haute performance des paquets réseaux par l'emploi de techniques comme le zero-copy, ou le kernel-bypass.

Title : System and Networking Aspects of the Transition of High-Performance Applications from Dedicated to Commodity Hardware: the Example of Media Production for Professional Broadcast

Keywords : media processing, packet pacing, jitter, instrumentation, commodity servers, software architecture

Abstract : Media production for professional broadcast (*i.e.,* the process by which multiple audiovisual sources are mixed and processed, in real-time, to elaborate the audiovisual stream as it will be consumed by the final viewer) has currently been implemented with dedicated hardware equipment, based on the Serial Digital Interface (SDI), an interconnection technology carrying the legacy of analog video. Despite an ongoing industrial effort to replace SDI with IP-based interconnection — as specified by the SMPTE 2022-6 and 2110 standards — the delay-sensitive nature of media production still challenges its total transition to software running on commodity servers. This thesis solves different aspects of that problem. First a quantitative and qualitative study of the sources of jitter undergone by those streams upon software processing has been conducted. In addition to results specific to Linux x86_64 servers, that work has yielded a general jitter exploration methodology, applicable to any operating system and hardware commodity servers. Second, a generic platform enabling the implementation of custom high-accuracy instrumentation for hardware-based packet timestamping has been developed. Despite being FPGA-based, that platform allows network and broadcast operators with little hardware design skills to specify custom logic for line-rate packet processing and timestamping, by using the P4 language. In particular, such instrumentation can be used to qualify the jitter properties of media production streams. Third, a system to perform packet-pacing — *i.e.,* the transmission of a constant-rate packet stream with negligible jitter — has been proposed. Despite exclusively relying on commodity hardware, the proposed system has been formally and experimentally proven to yield a jitter, conforming to the tight requirements of media production streams. Finally, a software framework easing the implementation of media-production applications has been developed. That framework relies on a separation between media processing and media transport. The transport layer handles high-performance packet processing with techniques such as zero-copy and kernel bypass networking.

Institut Polytechnique de Paris
91120 Palaiseau, France