MECaNIC: SmartNIC to Assist URLLC Processing in Multi-Access Edge Computing Platforms

Taejune Park*, Myoungsung You†, Jian Cui‡, Youngjin Jin†, Kilho Lee§, and Seungwon Shin†
*Chonnam National University, Gwangju, Republic of Korea
†KAIST, Daejeon, Republic of Korea
‡S2W Inc. Sungnam, Republic of Korea
§Soongsil University, Seoul, Republic of Korea

2022 IEEE 30th International Conference on Network Protocols (ICNP) | 978-1-6654-8234-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICNP55882.2022.9940263

***Abstract*—Multi-access edge computing (MEC) providing server capabilities at near end-users is introduced to enable Ultra Reliable Low Latency Communication (URLLC) for mission-critical and time-sensitive networked services. However, the current MEC simply shortens the physical travel distance of traffic but does not include any architectural approach for supporting URLLC. As a result, MEC implicates resource contention issues, and important packets can be easily delayed or lost, resulting in critical flaws for those services. To address these problems, we introduce *MECaNIC*, which extends the data plane of MEC to SmartNIC and assists URLLC of MEC. It provides i) precise packet scheduling that handles traffic priorities into two dimensions of reliability and latency, and ii) task offloading that accelerates MEC applications, including payload matching and response caching. The prototype implemented using NetFPGA shows that *MECaNIC* reduces the average latency of the high-priority traffic from 2,823 μs to 397 μs while ensuring packet delivery, even when the traffic competes with other lower priority traffic. Also, task offloading improves a MEC's payload processing 4-fold and reduces file downloading time and video random access time by 44% and 17%, respectively.**

***Index Terms*—Edge computing, URLLC, SmartNIC**

I. Introduction

Ultra low-latency and reliable communication (URLLC), which means under *1* ms latency and packet loss rates of *1:100,000* [1], [2], is considered to be the momentous requirement for mission-critical and time-sensitive networked services such as connected cars, Internet of Things (IoT), Cyber-Physical Systems (CPS), or multimedia as well as augmented/virtual reality (AR/VR) [3]–[13]. However, since it is challenging to meet the requirement of communicating with the traditional remote server capabilities (e.g., cloud), Multi-access edge computing (MEC) is emerged, which provides the server infrastructure at the edge of the backhaul network and in close proximity to end-devices/users in shortening the physical travel distance of network traffic. In this context, MEC should be able to handle various third-party services within the limited resource, so the structure of MEC is based on a virtualization environment (e.g., Network Function Virtualization) with COTS devices (e.g., x86 servers), and it is defined as the standard by European Telecommunication Standards Institute (ETSI) [14].

However, the following research question about the current MEC structure remains: *"MEC has been introduced for URLLC, but does MEC really support URLLC well?"*. Although MEC may be recognized as a new significant architecture to achieve URLLC, the current MEC simply follows a conventional virtualization system without any structural consideration for URLLC, except being physically near the end-hosts. In this regard, we found that several problems break URLLC in MEC. First, due to the characteristics of a virtualized environment, multiple traffic flows for several services are concentrated on a single MEC host, and various MEC services (applications) share limited hardware resources, making resource contention likely. This leads to non-deterministic results in traffic processing which can compromise reliable and low-latency communication, and even critical traffic (e.g., autonomous driving) can be delayed or dropped when they contend with other packets for less important services (e.g., video streaming). In addition, since network traffic arriving at MEC can be processed on a virtual machine only after going through a long network stack from a network interface to a kernel and a hypervisor, network latency inevitably increases and aggravates resource contention problems in packet processing. Lastly, the limited processing performance of software in packet processing becomes another bottleneck point of URLLC; services running on MEC usually operate on the L7 application layer, such as CoAP for IoT services [15], [16], HTTP for REST APIs and media contents [17], [18], or content offloading such as machine learning to MEC [19].

To properly handle requests on time, the performance of retrieving and analyzing packet payloads is an important feature for the MEC task. Unfortunately, pattern matching, a basic function for packet payload processing, is one of the most time-consuming tasks for software involving significant throughput degradation and latency increase [20], [21]. Consequently, although the primary goal of MEC is to achieve URLLC, we conclude that the current MEC architecture itself does not consider supporting URLLC ironically.

Several previous studies have suggested solutions to these problems. The resource contention issue has been studied at length with various works that propose resource sharing and scheduling methods [22], [23]. In addition, a number of studies [24]–[26] present L7 packet processing acceleration techniques via hardware offloading. However, we claim that these methods are not suitable for MEC and URLLC. While reliability and latency of URLLC are independent attributes

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

that should be managed separately, previous studies typically have grouped them together based on a single priority concept or have provided no further capabilities to facilitate URLLC than just faster packet processing.

To address these limitations, we conclude that the existing MEC architecture requires new URLLC-oriented packet processing schemes: 1) a new scheduling system that supports application layer protocols and manages packet priorities with regard to reliability and latency separately, and 2) a MEC application offloading system that mitigates the computation overhead of packet processing tasks. To enable them on a MEC host, we propose *MECaNIC* that extends the data plane of the MEC architecture into hardware (i.e., network interface card, NIC) and supports URLLC through fine-grained scheduling and task offloading. Then, *MECaNIC assists* the MEC host at the network-level to enable stable and reliable URLLC networking and service processing. By leveraging the responsiveness and parallelism of hardware, *MECaNIC* determines packet priorities as soon as network interfaces receive each packet, even taking payload into account and provides precise scheduling that can satisfy both reliability and latency requirements of each network flow. Also, *MECaNIC* can perform a part of application tasks before forwarding the packets to the host layer, such as pattern matching or responding to a specific request instead of MEC applications. This *task offloading* can mitigate processing delays of time-consuming tasks and reduce overhead made by the MEC applications, improving the overall response time of MEC.

We implement a *MECaNIC* prototype using NetFPGA-SUME [27], and our evaluation shows that *MECaNIC* supports precise scheduling while considering L7 protocols. *MECaNIC* significantly reduces and stabilizes the average latency of high priority traffic (i.e., from 2,823 μs to 397 μs) while maintaining reliable communication even under a contention with low priority traffic. Moreover, *MECaNIC* significantly improves the performance of MEC applications through task offloading. It increases MEC's payload matching throughput by 4-fold, and it reduces file downloading time and video random access time by 44% and 17%, respectively.

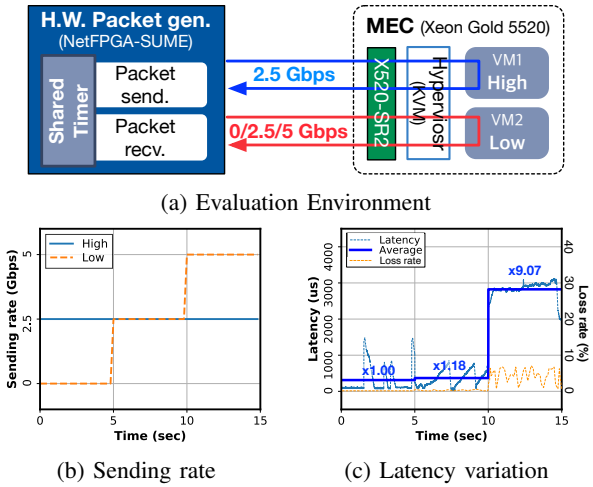
II. Background and Motivation

A. URLLC and MEC

Ultra-reliable low-latency communication (URLLC), which supports sub-*1 ms* latency with packet loss rates of *1:100,000*, is a critical requirement for on mission-critical or time-sensitive services [1], [2]. To achieve this network requirement, multi-access edge computing (MEC), which places service infrastructure on behalf of remote clouds at or near end-users, is considered a pivot architecture by reducing the physical travel distance of network traffic [14], [28], [29]. The standard MEC architecture defined by the European Telecommunication Standards Institute (ETSI) is based on virtualization techniques with COTS devices (e.g., x86 servers) to allow a multitude of third-party services on MEC as cloud infrastructure, i.e., Platform-as-a-Service (PaaS). [19].

Reliability requirement	Latency requirement	Examples
Sensitive	Sensitive	Emergency brakes, Medical alarms
Sensitive	Tolerant	Healthcare, File downloads
Tolerant	Sensitive	Multimedia streaming, AR/VR
Tolerant	Tolerant	Common web services, best effort traffic

TABLE I: Reliability and latency requirements of MEC traffic



(a) Evaluation Environment

(b) Sending rate

(c) Latency variation

Fig. 1: MEC measurement environment

Traffic model: URLLC not only enables responsive low-latency communication, but also supports stable and reliable communication. We argue that these two factors of communication, *reliability* and *latency*, should be treated as orthogonal elements. Thus, we should consider MEC traffic that requires URLLC into four classes as presented in Table I. As described in the table, safety-critical applications, such as medical alarms, require reliable and latency-sensitive communication. Some applications are reliability-sensitive but latency-tolerant. For instance, a healthcare that collects information from IoT devices should reliably track the health conditions but does not always have to be updated quickly. On the other hand, video streaming services may tolerate some packet losses, while latency is a key factor for QoS. Therefore, it is essential for a MEC host to precisely consider the different reliability and latency requirements for each application to bring high-quality services to end-users.

B. Problem: Failure of URLLC on MEC

However, the current MEC architecture is practically a common NFV system and does not have any particular structure or feature for URLLC other than simply reducing the physical distance. That is, although MEC is introduced for URLLC, it may be difficult to say that MEC complements URLLC well. We find that the existing MEC architecture can compromise URLLC: 1) The latency exceeds 1 ms and 2) the packet loss rate increases more than 1:100,000 like the following:

Motivating example: Fig. 1a shows a motivating example of the failure of the URLLC communication in a test environment. The MEC host is leveraged with Intel Xeon Gold 5520, 64GB of RAM and two Intel X520-SR2 NICs. It runs two virtual machines on top of the KVM hypervisor [30]

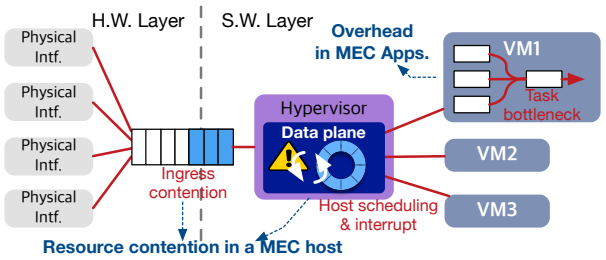


Fig. 2: Problems that hinder URLLC for MEC systems

that respectively serve safety-critical and best-effort services (e.g., VM1 for vehicle control and VM2 for infotainment). Incoming packets to NICs are forwarded to the destination VM while passing through the hypervisor. In this environment, we concurrently send two traffic flows: VM1 (i.e., Noted as *High*, meaning high priority) at 2.5 Gbps for 15 seconds and VM2 (i.e., Noted as *Low*, meaning low priority) at 0, 2.5 and 5 Gbps every 5 seconds (See Fig. 1b). For precise measurement, we utilize a hardware packet generator implemented with NetFPGA-SUME, an FPGA-based PCI express 10 GbE NIC [27]. It transmits a certain amount of packets to each interface at a specified rate and measures its round-trip time with a precision of 6.25 ns [31].

Result: As seen in Fig. 1c, the latency of the *high* traffic is significantly affected by the *low* traffic, as the *low* increases its sending rate. The average latency increases 9-fold from 311.95 to 2823.17 μs. Also, the loss rate increases up to 7%, meaning a loss of 14,000 packets per second at 2.5 Gbps, which far exceeds the reliability requirement of 1:100,000. One may claim that this performance degradation seems minor, but it leads to failures in providing URLLC for time and mission-critical services. For instance, consider the *high* traffic as traffic for autonomous driving control, which requires the latency of less than 1 ms with high-reliability for safety [32]–[34]. In this case, packets delayed by 3 ms with some loss may be critical for a braking system to cause an accident.

Reason: We precisely analyze the result and found that the main reasons for this problem are *resource contention* in the MEC host and *overhead* in the MEC VM/applications (Fig. 2). When multiple MEC applications are executed simultaneously on a single MEC host, resource contention at the MEC host is inevitable. This is because the MEC host consists of a conventional network function virtualization architecture. It provides software abstraction for an isolated network channel, while sharing underlying hardware resources, such as network fabric, caches, buses (i.e., PCI-E), and memory controllers [23], [35]. Despite the contention, the data plane of the MEC host simply handles traffic in an FCFS (First Come First Served) manner. With FCFS, packets could be lost or delayed randomly according to their arrival order, because each traffic inherently contains some jitters and bursts. Another reason is the overhead in MEC applications. When MEC application (VM) handles reached packets, it takes computation time to process them. Also, according to VM scheduling, packet processing could be delayed even if all packets arrive. Such overhead can impose large delays and packet losses.

Research	Reliability scheduling	Latency scheduling	L7-aware scheduling	Packet accel.
HUG '16 [22]	●	✖	✖	-
PARTIES '19 [43]	✖	●	✖	-
PicNIC '19 [23]	●	●	✖	-
FlowBlaze [24]	-	-	●	●
AccelTCP '20 [26]	-	-	●	●
PANIC '20 [44]	●	●	✖	●
FairNIC '20 [45]	●	●	✖	●
Vajiheh '19 [36]	●	✖	✖	-
TODG '22 [37]	●	●	✖	-
Tianle '21 [46]	✖	✖	✖	●
MECaNIC	●	●	●	●

TABLE II: Prior studies on scheduling and packet accel.

Implication: These problems of URLLC on MEC can easily occur at any time considering the operating environment of the MEC; MEC has to deal with multiple traffic flows for different services in the VM-based environment, thus traffic contention and resource contention are frequent. Also, as MEC typically deals with application services, such as IoT, automotive, VR/AR and computation offloading to MEC [36], [37], its network traffic is commonly delivered over application layer protocols (e.g., CoAP [15], [16] and HTTP [17], [18]). For example, most IoT services adopt HTTP REST API to provide diverse functionalities in an efficient and unified manner (e.g., On a medical IoT service, to query patient info, `GET api/monitor/patient/[name]`) [38]–[40]. In this context, to identify and process services requested by users, MEC VMs need to consider application-layer contexts (e.g., REST API URLs and methods) by performing pattern/string matching on the packet's payload. However, pattern matching is known to be a time-consuming task, which means not only is this operation itself a loss in latency, but it can also cause resource contention with other VMs [41], [42]. Consequently, these issues can cause fatal errors for mission-critical and time-sensitive services as we mentioned above.

C. Related work

To address the above problems, traffic scheduling to avoid the contentions and offloading to accelerate the rate of processing have been considered in many prior studies as summarized in Table II. HUG [22] proposes bandwidth isolation by resource fairness. PARTIES [43] mainly focuses on guaranteeing low-latency by resource partitioning, and PicNIC [23] focuses on end-to-end performance isolation by resource sharing and an admission control. They may be help to avoid the resource contention issue by network scheduling. In terms of packet processing acceleration, FlowBlaze [24], SmartTLS [25], and AccelTCP [26] accelerate network processing by offloading some network features into network interface cards (NIC). In addition to the acceleration, PANIC [44] and FairNIC [45] manage packet scheduling together. These studies might help MEC in addressing the problems for supporting URLLC. However, while the two critical factors in URLLC come from the reliability and latency in processing, most of those solutions have considered only one of the two aspects. Moreover, the few studies that did consider the two factors are still not tailored to URLLC as the application layer is disregarded.

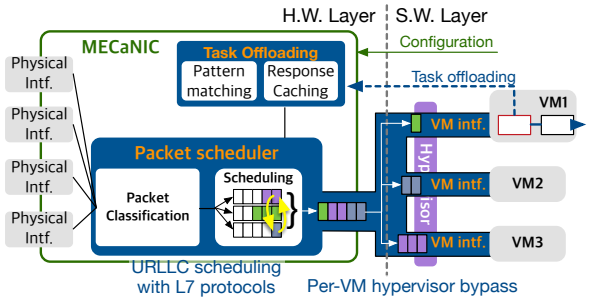


Fig. 3: *MECaNIC* overall design

The characteristic of URLLC is not much considered even in the the studies on MEC. Vajiheh [36] and TODG [37] suggest effective computation offloading (i.e., end-devices to MEC) strategies, but the side-effect on the MEC host by the offloading is not considered. Tianle [46] proposes a way to accelerate MEC with hardware, but the reliability for URLLC is not invested. Other researches on MEC focus on improving the responsiveness of the network or providing an optimized network fabric, such as optimal MEC VM / application placement in a network [47], [48], lightweight MEC service initiation for responsiveness [49]–[51], or efficient service utilization with MEC [52], [53], and overlook the network architectural issues in which URLLC can be compromised.

D. Research goal

To support URLLC on MEC while addressing the limitations of previous studies, we conclude that the MEC architecture can be improved by designing a smartNIC-based MEC data plane to assist URLLC networking of MEC. This aims to achieve 1) new packet scheduling that manages traffic flows with different requirements (Recall Table I) at the same time, and 2) task offloading that assists proactive packet processing or message processing without host software intervention. As a result, the reliability and latency aspects of traffic can be controlled independently and alleviates MEC application overhead, resulting in stable URLLC processing on MEC.

III. System Design

Based on our insights, we propose a novel hardware-based traffic management system '*MECaNIC*', a SmartNIC residing between the host layer and network interfaces.

Design constraints: In a PaaS model, developers could write custom SmartNIC applications and submit them to a platform operator for approval and deployment onto SmartNICs. In this context, since *MECaNIC* is designed as a SmartNIC for the MEC architecture operating as a PaaS, we assume that tenants exploit *MECaNIC* through its configuration APIs, and do not attempt to circumvent our isolation mechanisms [45].

A. Architecture overview

As shown in Fig. 3, *MECaNIC* consists of two major components: *MECaNIC-scheduler* and *MECaNIC-task offloading*. *MECaNIC*-scheduler is a new queueing architecture that can achieve reliable bandwidth reservation and ultra-low latency communication concurrently. *MECaNIC*-task offloading consists of hardware-implemented network functions to accelerate MEC applications in terms of throughput and latency.

ID	Match		Priority			Actions
	Header	Keyword	Queue	Reliability (P^r)	Latency (P^l)	
1	A → B	-	0:3	10	10	Fwd(App1)
2	A → B	"Urgent"(12,6)	0:3	50	100	Fwd(App1)
3	C → D	"Request"(24,8)	4:9	5	10	Payload(10), Fwd(App2)
4	E → F	"GetInfo"(12,8)	0:9	1	10	Cache(10)

TABLE III: Packet classification table

***MECaNIC*-scheduler (§III-B and III-C):** Upon receiving each packet, the *packet classifier* enqueues the packet into the *MECaNIC-queue* according to the packet classification table (See Table III). The table specifies the packet matching entries with the priority of reliability / latency requirements (See §III-B for more details). *MECaNIC*-queue also consists of multiple queues (i.e., shared queue pool) and provides *preemptive queue allocation* and *per-packet priority queueing*. For reliability, each flow can allocate several queues in the pool. The key idea is that a flow with a higher reliability priority can preempt queues allocated to other flows with lower reliability priorities. With this, the queue pool can provide bandwidth reservation by dequeuing packets in a round-robin (RR) manner. However, since RR always fairly shares the bandwidth with other flows, it is not suitable for supporting ultra-low latency requirements. To address this problem, *MECaNIC*-queue schedules packets in per-packet manner in order to priority them accordingly. Note that the dequeue priority is determined by the latency requirement of each flow. With this, *MECaNIC* can schedule latency-sensitive traffic first, even if it makes some jitters and bursts. However, realizing such the queuing algorithm is challenging, because it may incur heavy overhead in managing queues, especially when updating each packet's priority value. To resolve this issue, *MECaNIC* fully takes advantage of a hardware-based system; it can concurrently execute major queue operations, including priority value updates and queue lookups. In addition, *MECaNIC* provides per-VM hypervisor bypass to avoid overhead from the hypervisor's networking stack.

***MECaNIC*-task offloading (§III-D):** MEC VM / applications can offload some operations into *MECaNIC* so that packets can be processed before arriving at the VM. It helps to improve throughput and latency at the same time through alleviating computation and internal communication overheads. *MECaNIC* provides hardware-implemented network functions that support: 1) payload matching, a widely used compute-intensive network function, and 2) response caching that can significantly reduce internal communication overhead. The main idea of the accelerator is to utilize the parallelism and responsiveness of the hardware-based architecture.

Control interface: *MECaNIC* aims to act as a kind of SmartNIC that works by binding to a host machine and providing additional functions more than the packet transmission / reception. In order to effectively communicate with the host machine, *MECaNIC* has its own device driver on the host software layer and provides a set of APIs for management.

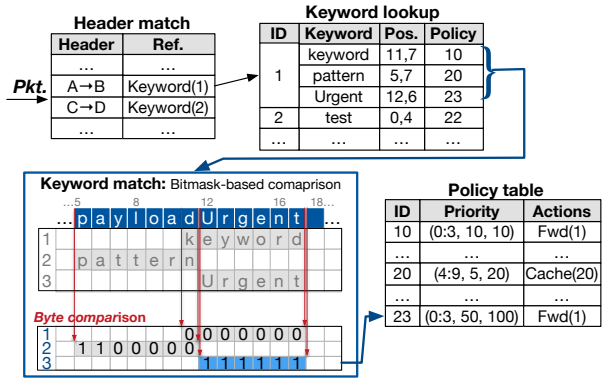


Fig. 4: Packet classification

From this host control interface, the host (or its network administrator) is able to define packet priority or offloading for incoming packets by configuring tables and other settings of *MECaNIC* as necessary.

B. MECaNIC-scheduler: packet classifier

Incoming packets to *MECaNIC* undergo a classification process to identify the corresponding priority and actions. This lookup operation is performed to match packet headers (e.g., 5-tuple), and the *keywords* that search a certain keyword in the specified position and length $(position, len)$ in the packet payload by bitmask operation. Fig. 4 describes its processing steps: A packet is first parsed and matched with the header table and then matched to a set of keywords from the keyword table. The keyword matching process looks packet bytestream up keywords in the table byte-to-byte. A keyword match is found by performing an AND operation on the keyword arrays and the payload's byte stream, thereby determining packet priority and actions. For example, in Fig. 4, the payload is matched to the "Urgent" keyword in 12-17 byte location of the third array, and the policy with ID 23 ($\langle$ *(0:3, 50, 100), Fwd(1)* $\rangle$) is assigned to the packet. The queuing / scheduling process of this policy is described in the following section.

This keyword processing model is elaborately designed and required to classify L7 protocols in low-latency: First, since keyword search is performed by specifying the location for comparison, the operation can be executed immediately without the need to read the payload sequentially. Second, since the hardware is specialized for parallel processing, matching is completed within a single clock operation no matter the length or the number of keywords. Therefore, *MECaNIC* can process multiple keywords at once with tiny overhead. Using this property, even if the position of a specific keyword in the payload is variable, it is possible to handle comparisons by matching multiple identical keywords by changing positions within the variable range. General pattern matching that searches for an unknown position is supported in the process offloading module, described in §III-D.

C. MECaNIC-scheduler: queueing

Queuing/scheduling model: After packet classification, the *MECaNIC*-scheduler enqueues the packet into the *MECaNIC*-queue that provides an advanced queueing mechanism for URLLC. *MECaNIC*-queue consists of a number of queues (i.e., shared queue pool) and provides *preemptive queue allocation* and *per-packet priority queueing*. The basic principle is that reliability-sensitive traffic has higher priority to prevent packet loss by guaranteeing its own bandwidth, and latency-sensitive traffic has higher priority for earlier processing.

For reliability, *MECaNIC* provides a preemptive queue allocation mechanism that ensures a flow with a higher reliability level not being lost by bursts of lower reliability flows. It reserves the bandwidth of each flow by specifying a queue range *Queue*$(i : j)$ of the shared pool. Here, the queue range can be allocated even if it overlaps with ranges of the other flows, but flows having higher reliability priority P_f^r can preempt queues assigned to the other flows that have lower priority P_f^r. By default, the *MECaNIC*-scheduler dequeues packets in a round-robin manner. Then, each queue in the pool can reserve a certain amount of bandwidth (i.e., $\frac{total\ bandwidth}{the\ number\ of\ queues}$), and the flow f with higher P_f^r can have a more reliable bandwidth reservation.

To provide low-latency communication while preserving bandwidth allocation, *MECaNIC* provides per-packet priority queueing that makes ensures a flow with a higher latency level being out the queues earlier than lower ones. *MECaNIC* assigns a *dequeue priority* value dp_p for each packet p. As each packet is generated by a flow, dp_p is initialized based on the latency priority P_f^l of the flow, and is increased over time while packet p waits in queue to avoid starvation. Note that the higher the latency priority P_f^l, the highest the dequeue priority dp. Consequently, the highest latency priority packet forwards earliest to the host layer by *MECaNIC*.

We believe that this per-packet priority model is quite an effective scheduling strategy to the URLLC scale because even the same flow can precisely control the packet processing order that can vary depending on payload contexts. Despite this advantage, one may be concerned with the overhead of per-packet priority as it may require more computation compared to the conventional priority queue which assigns a static per-queue priority. However, *MECaNIC* is designed to take advantage of parallelism in a hardware-based system; it can update and compare the priority information of every packet simultaneously.

Scheduling operation: Algorithm 1 describes the details of scheduling. As stated above, *MECaNIC*-queue is implemented by exploiting the hardware parallelism to process multiple queues in parallel. It consists of two steps: *enqueuing* packets into the shared queue pool, and *dequeuing* packets according to their priority. Scheduled packets are forwarded to the host layer in turn. Upon receiving a packet from network interfaces, *MECaNIC* collects packet body (i.e., in_packet) and scheduling parameters including the queue range ($[i : j]$), reliability priority (P_f^r), and latency priority (P_f^l).

For enqueueing, *MECaNIC* then gets the candidate queue list $Candidates$ from the queue pool $QueuePool$ with the queue range $[i : j]$ (Line 2). In Line 3, *MECaNIC* gets queues that can be allocated to the given packet ($Alloc$).

Algorithm 1: *MECaNIC* scheduler queueing

Input: in_packet, rule $([i:j], P_f^r, P_f^l)$

1 $>>$ *Upon receiving every packet, enque*
2 $Candidates \leftarrow QueuePool[i:j]$
3 $Alloc \leftarrow \{q \in Candidates |\ q.P^r \leq P_f^r\}$
4 $Lowest \leftarrow \{q \in Alloc |\ q.P^r$ **equals** $min_{q \in Alloc}(q.P^r)\}$
5 $q \leftarrow argmin_{q \in Lowest}(q.ts_{last\_in})$
6 $q.P^r \leftarrow P_f^r$
7 Enqueue $(q, pkt(in\_packet, P_f^l, P_f^l, clk))$
8 $\Rightarrow$ `pkt:(in_packet, dp, dp_tick, in_time)`

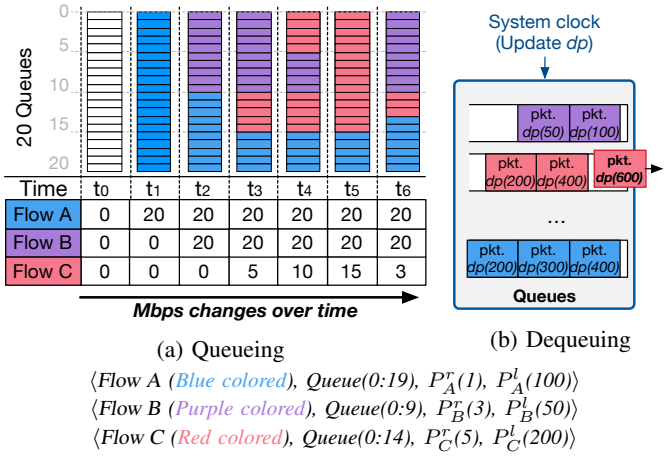
Output: *out_packet*

9 $>>$ *At every time tick, dequeue*
10 $targetq \leftarrow argmax_{q \in QueuePool[:]}(q.head.dp)$
11 **if** $\exists targetq$ **then**
12 $pkt \leftarrow targetq.dequeue()$
13 $out\_packet \leftarrow pkt.in\_packet$
14 Forward out_packet to the host layer
15 **end if**
16 **parfor in** $q \in QueuePool[:]$
17 **parfor in** $pkt \in q$
18 $pkt.dp \leftarrow pkt.dp + pkt.dp\_tick$
19 **end parfor**
20 **end parfor**

Note that $q.P^r$ inherits the last enqueued packet's P_f^r value. Therefore, the packet can be allocated to queues for lower or equal priority flows (i.e., $q.P^r \leq P_f^r$). In Line 4, *MECaNIC* gets queues having the least $q.P^r$ value. It then picks the oldest used queue q as the target queue. Note that $q.ts_{last\_in}$ contains the timestamp at the last packet enqueued. After that, *MECaNIC* updates $q.P^r$ with the current packet's value (Line 6), and enqueues the packet (in_packet) into the target queue q. For further scheduling, packets are enqueued with additional information, P_f^l, P_f^l, and clk as an initial dequeue priority (dp), a priority update value per-tick (dp_tick), an enqueued time stamp (in_time), respectively.

For dequeueing, at every device clock cycle, *MECaNIC* picks $targetq$ that has the packet with the largest dequeue priority in its head (Line 10). Note that $q.head.dp$ represents the dequeue priority (dp) of the packet in the queue head ($q.head$). If *MECaNIC* gets the target queue that contains such a packet, it then dequeues that packet and forwards it to the host (Line 11-15). After that, *MECaNIC* increases dp values of all packets in the queue pool by dp_tick (Line 16-20). Note that our *MECaNIC* prototype implements the algorithm while fully utilizing hardware parallelism. For instance, it updates all packet information at the same time (Line 16-20).

Operation example: Fig. 5a describes how *MECaNIC*-queue works with three flows having different scheduling requirements. For ease of presentation, we assume that the queue pool has 20 queues and the reserved bandwidth of each queue is 1Mbps (i.e., 20 Mbps of total bandwidth). For bandwidth allocation, Flows A, B and C are allocated with queue-range of 0:19, 0:9 and 0:14, respectively. Note that Flow C has the highest reliability priority, and Flow B and A have the second and third highest, respectively ($P_C^r > P_B^r > P_A^r$). At the beginning (t_0) of enqueueing, all queues are empty. At t_1, queues [0:19] are allocated to Flow A. At t_2, Flow B starts



(a) Queueing (b) Dequeuing

⟨*Flow A (Blue colored), Queue(0:19),* $P_A^r(1)$, $P_A^l(100)$⟩
⟨*Flow B (Purple colored), Queue(0:9),* $P_B^r(3)$, $P_B^l(50)$⟩
⟨*Flow C (Red colored), Queue(0:14),* $P_C^r(5)$, $P_C^l(200)$⟩

Fig. 5: Operation example of *MECaNIC* queueing system

to send traffic and preempts queues [0:9], because $P_B^r > P_A^r$. Note that although the sending rate of Flow B is 20 Mbps, its actual bandwidth is limited to 10 Mbps as the queue range is set to 10 queues. From t_3 to t_6, queues are allocated to Flow C which has the highest P^r. At t_6, as the sending rate of Flow C decreases to 3 Mbps, some queues occupied by Flow C are released, and Flow A and B can use them again. Fig. 5b depicts the dequeue process. *MECaNIC* dequeues a packet with the highest dequeue priority value (dp_p) first. In the figure, Flow C's packet that has the highest latency priority P^l is dequeued first when it competes with other flows.

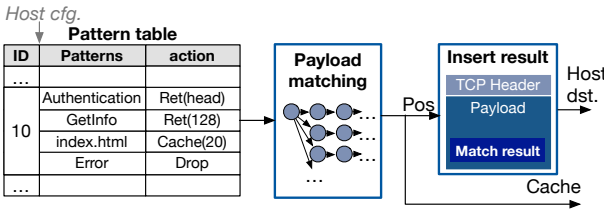
We can consider the previous example as a scenario for an autonomous vehicle [3]. Flow A, B, and C could be ambient environment monitoring, cruise controlling, and emergency braking, respectively. While driving, bandwidth is appropriately distributed to all flows. However, in a dangerous situation, emergency braking (Flow C) can be supported through reliable and low-latency communication by prioritizing it over all other flows, and the vehicle can preserve its safety.

Per-VM hypervisor bypass: Even if the data plane well prioritizes incoming flows, they may suffer from additional delays and packet losses, because a hypervisor typically employs a long software-based networking stack which makes packet buffering and contention unpredictable. To avoid this uncertainty, as shown in Fig. 3, *MECaNIC* introduces the hypervisor bypass interface which directly binds *MECaNIC*'s interfaces to each VM, enabling packets to arrive at each VM's virtual network interface while bypassing the unpredictable hypervisor network stack.

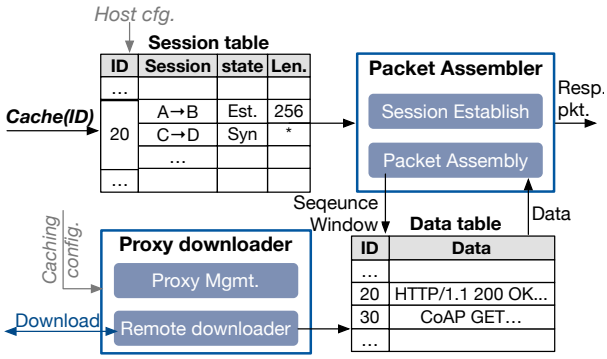
D. MECaNIC-task offloading

To accelerate heavy tasks that adversely affect the overall performance on the MEC host, the task offloading module pre-processes packets instead of the MEC host applications.

Payload matching: To parse payload for L7 protocols, payload matching is one of the indispensable features in network processing. Unfortunately, it is also known as one of the most expensive operations that delay processing time and decrease the throughput of network applications [54], [55].

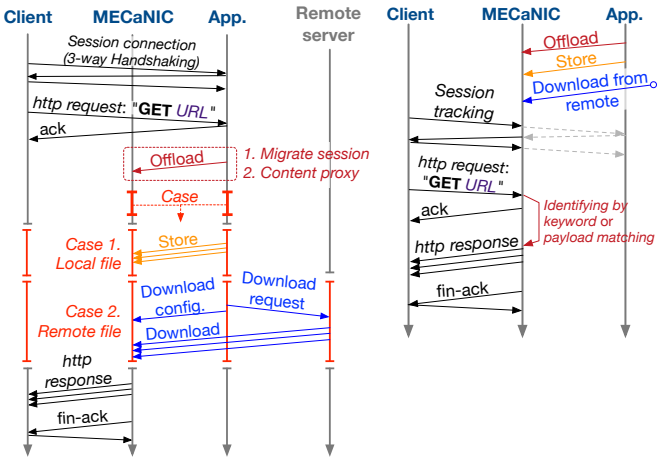


(a) Payload matching module



(b) Content caching module

Fig. 6: Design of task offloading module



(a) By session migration (b) By pre-fetched message

Fig. 7: Example of the response caching for HTTP

MECaNIC can pre-process payload matching before forwarding packets to MEC VM / applications. Thus the application recognizes the expected payload as soon as packets are received, drops unnecessary packets directly at the NIC, or calls response caching so that *MECaNIC* responds instead. Fig. 6a describes its structure and operation: Patterns to find in a packet's payload and actions to be taken when detecting the patterns are stored in the pattern table with a payload ID. When the *actions* 'payload()' is called with the ID at the classification table, corresponding patterns of the ID in the pattern table are loaded into a set of linear state machines, which sequentially enters each byte of packet payload into the state machine on every clock cycle, looking for matching patterns inside the payload. If a pattern match occurs, *MECaNIC* will perform the actions for each pattern: i) If the matching result should be returned to an application, *Ret* action is set, and the matching result (i.e., matched pattern and its position) is enclosed in a designated position in the packet payload such as payload head, tail, or an arbitrary location set in advance by an application (e.g., *Ret(128)* refers to the 128 bytes offset in the payload). Therefore, the application can proceed immediately to the main procedure related to packet processing by referring to the designated field without needing extra time to parse the packet; ii) The *response cache* (To be explained soon) is called to instead send a response message pre-allocated into *MECaNIC*; or iii) *drop* the packet at the NIC directly.

Response caching: In network services, performing a response to redundant requests every time causes considerable overhead. In particular, frequent responses or long-lasting network sessions (e.g., video streaming services) [8] should require reasonable network bandwidth for each session to guarantee reliable network services, which can incur network overhead in the MEC host [56]. Moreover, when MEC provides a proxy service that downloads content from a remote cloud and delivers it to end-devices, it requires double the bandwidth, resulting in contention between the traffic by itself, significantly reducing overall stream performance.

To reduce latency and overhead in data transmission on MEC VM/applications, *MECaNIC* provides response caching that allows a host application to pre-fetch response messages into *MECaNIC* to make responses for specific requests at the data plane level directly. Fig. 6b shows the overall response caching design. A host can store some messages to be cached per requests into a *MECaNIC* memory (i.e., Data table in the figure) by deploying rules with a *cache ID*, and these cached messages are managed by *MECaNIC* until the rules are deleted. Here, the messages can be stored in two ways: i) A host directly stores response messages into *MECaNIC*, i.e., *MECaNIC* acts like a caching server. ii) A host can set on a download address of an outside (e.g., remote cloud) and directly fetch and store data from that address, i.e., *MECaNIC* acts like a content proxy server.

The cached message is assembled as response packets and delivered to clients over TCP, so *MECaNIC* manages its own TCP session table to keep TCP sessions of requests for the cached data in two ways: *i) Identifying requests on a host but migrating the data transfer on MECaNIC.* After identifying request messages, a host application can delegate tasks that are expected to have significant data transfer overhead into *MECaNIC*. This operation is achieved by *session migration* from a host by copying the corresponding session status (i.e., 5-tuple, states, and SYN/ACK numbers) into *MECaNIC*'s session table. Fig. 7a shows the example of this operation for a *http* request to a service application. The application establishes a session in the usual way and prepares to send content according to the 'GET *URL*' request. Here, the application can migrate the TCP session about the request into *MECaNIC* with the response message from the application or the remote server. Finally, the message is sent to the client as assembled packets without the application's intervention.

ii) Identifying requests AND transferring responses on MECaNIC. When a host application deploys a rule with a response message for a specific request in advance, *MECaNIC* can identify the request with the keyword or payload matching while tracking its TCP session and make the response directly. Fig. 7b is an example of this operation. As some request messages with their responses can be specified in advance, the application can pre-fetch the responses into *MECaNIC* from the application or the remote server. Then, when a specific request is detected, *MECaNIC* immediately assembles the pre-fetched data into the response packets and transfers them to the client on behalf of the application.

IV. Implementation

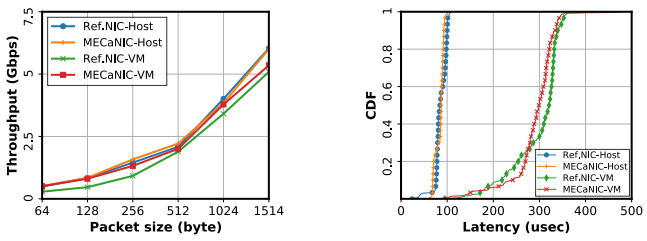
To validate the design of *MECaNIC*, we have implemented a *MECaNIC* prototype with NetFPGA-SUME, an FPGA-based network card [27]. All FPGA codes are written in Verilog and synthesized with Xilinx Vivado. The prototype is deployed into a server with Intel Zeon Gold 5520@2.2GHz, 64GB memory, and Ubuntu 16.04. The device driver and APIs are extended from the reference driver of NetFPGA-SUME.

There are several constants to determine the scalability of *MECaNIC*. Note that these constants in this implementation are values set for rapid prototyping and can be modified as needed in the future: 1) The packet control module is set to manage 1,024 5-tuple entries, and keyword matching is configured to support 100 keywords per keyword ID of the same header; 2) The priority control module is implemented with 20 queues and can allocate up to 500 Mbps of bandwidth per queue depending on its criticality; and 3) The interface control manages 16 queues so that the *MECaNIC* prototype can support 16 VMs. Payload processing is implemented with a state machine to handle 100 patterns of 128 bytes per pattern ID concurrently, and matching is performed by driving the state machines with shift-registers for incoming packets. The response caching uses DDR3 memory equipped in NetFPGA-SUME and manages up to 4GB of data.

Supporting encryption: Processing encrypted traffic on *MECaNIC* is allowed in a similar manner to other common SmartNICs; Most SmartNICs support TLS/SSL offloading by providing a way for VMs (or applications) to encrypt the packet using SmartNICs' crypto module, and processing encrypted traffic with such modules is a very typical method when SmartNICs are considered [25], [57], [58]. The same method can be adopted on *MECaNIC* for handling encrypted traffic in the future. In this context, an encryption key is held by each VM and sent to *MECaNIC* through its configuration APIs, and *MECaNIC*'s hardware and device driver can verify and restrict VMs to access the relevant keys only.

V. Evaluation

Evaluation environment: The test environment consists of the MEC host and the traffic generator, similar to Fig. 1a; The MEC host has an Intel Xeon Gold 5520, 64 GB of RAM and NetFPGA-SUME for the *MECaNIC* prototype. It runs virtual machines on the KVM [30] and Open vSwitch



(a) Throughput (b) Latency

Fig. 8: Performance baseline of NetFPGA and *MECaNIC*

[59], [60]. We measure network performance with/without *MECaNIC* in certain situations, such as giving priority to the VMs or running applications on the VMs. The test traffic is generated from the hardware packet generator used in the motivating example that transmits a certain amount of packets at a specified rate and measures its round-trip time with a precision of 6.25 ns.

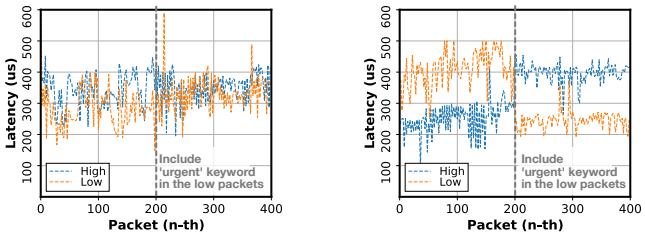
Performance baseline: Please note that the main goal of *MECaNIC* is to achieve reliable communication by precise scheduling and bandwidth allocation for the low latency services on the MEC host, *not* to improve the absolute throughput or to reduce the overall latency. Hence, before proceeding evaluations, we set the performance baseline to figure out the architectural overhead added by *MECaNIC*. For this, we first measured the performance of vanilla NetFPGA-SUME via the reference NIC of NetFPGA-SUME, which is the typical NIC designed by the NetFPGA official group so that can be considered an objective performance comparison target for NetFPGA-SUME implementations [61] Then, we compared the performance with *MECaNIC*.

In Fig. 8, `Ref.NIC-*` shows the throughput and latency results of the reference NIC when bursty traffic passes through the hypervisor only (i.e., host only) or a VM that returns incoming packets immediately without doing anything. Overall performance is converged to 5 and 6.5 Gbps and 100 and 350 μs respectively in each case, and they can be considered as our performance baseline. Comparing this result with *MECaNIC* (i.e., `MECaNIC-*` in Fig 8), *MECaNIC* has very similar performance to the reference NIC, i.e., there is no performance degradation by *MECaNIC*.

According to our analysis, the performance is mostly constrained by the device driver of NetFPGA and the network processing on host software and VMs, so they vary on what services run on. Therefore, our evaluations mainly focused on how helpful *MECaNIC* is to a MEC host processing compared to the reference NIC under various conditions and services, and clear that this does not represent the absolute performance limit of *MECaNIC*. In this context, we mark that there is room for further improvement in absolute performance by improving the device driver and the host system in the future.

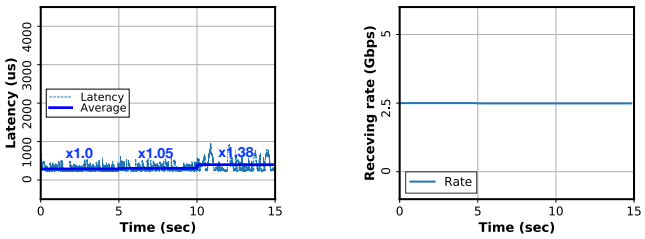
A. Scheduling measurement

Packet scheduling: *MECaNIC* can schedule the processing order between packets by comparing their priorities. To evaluate its packet scheduling, we send two request packets



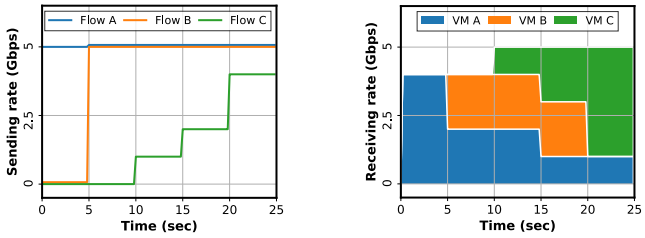
(a) Without *MECaNIC* (b) With *MECaNIC*

Fig. 9: Scheduling for two packets (high/low priorities)



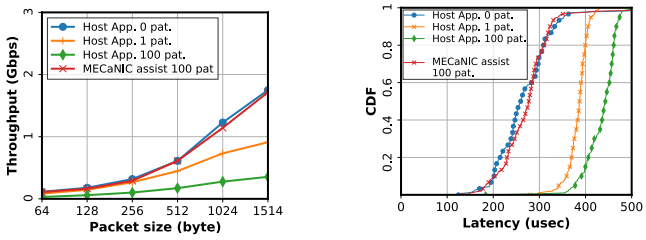
(a) Latency variation (b) Receiving rate variation

Fig. 10: Performance variations of the high-traffic under the contention condition (See with Fig. 1)



(a) Sending rate (b) Receiving rate

Fig. 11: Bandwidth reservation



(a) Throughput (b) Latency

Fig. 12: Performance of pattern processing

of different priorities, high and low, to each VM at the exact same time using the hardware packet generator, and measure their response times. A total of 400 packets are transmitted, but starting from the 200th packet, an 'urgent' message is included in the payload of the low-packets so that low-packets take higher priority. The packet interval is set to 1 sec.

Fig. 9 shows the result. Without *MECaNIC*, packet response times are disordered regardless of priority, and it is difficult to discern which packets are preferential even when the low-packets are specified with 'urgent' messages. In contrast, with *MECaNIC*, the high-packets are definitely processed earlier than the low-packets according to their priorities, resulting in faster response times. Also, *MECaNIC* is able to determine packet priority while considering payload (i.e., L7 protocol) by keyword matching. Therefore, the 'urgent' message included in the low-packets is identified so that *MECaNIC* allows low-packets to be processed earlier than the high-packets.

Under contention: To validate that *MECaNIC* can address the reliability issue in the case of contention, we performed the same evaluation of the motivating example in Fig. 1 with *MECaNIC*. We measured the changes in latency and throughput of the high-traffic delivered at a rate of 2.5 Gbps while gradually increasing the sending rate of low-traffic for contention from 0, 2.5 and 5 Gbps every 5 sec.

Fig. 10 shows its result. The average latency increase is only x1.38 (287.44 $\rightarrow$ 397.78 μs), a significant reduction from x9.07 of the motivating example. Also, as latency becomes relatively stable, the overall latency is slightly reduced. In addition, the throughput remains constant regardless of the increase in low-traffic, i.e., the loss rate is decreased to zero.

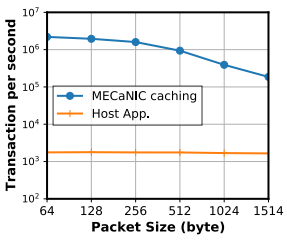
The slight latency variation (0.38) in the 10-15 sec period (i.e., the low-traffic is at 5 Gbps) is caused by processing delay for the low-traffic in the host; Since low-traffic carries twice as many packets as high-traffic in that period, there is a moment where the host receives and processes low-traffic only. At this moment, the next high-traffic packet needs to wait. This may be alleviated if the host process scheduling works in cooperation with *MECaNIC*.

Bandwidth control: *MECaNIC* can control bandwidth allocation by the queue-range bound. To validate this, we send three flows toward three VMs (i.e., Flow A, B and C), and each flow entry is set to share 5 Gbps of bandwidth; For the queue range of 0-9 for a 5 Gbps capacity (i.e., 0.5 Gbps per queue), the entries are set to $\langle$ *Flow A, Queue(2:9), Priority(1)* $\rangle$, $\langle$ *Flow B, Queue(2:5), Priority(3)* $\rangle$, $\langle$ *Flow C, Queue(0:7), Priority(5)* $\rangle$. That is, each flow is allocated with 4, 2 and 4 Gbps of bandwidth, respectively, but the queue ranges are slightly different. Then, the flows are generated while changing the sending rate as shown in Fig. 11a.

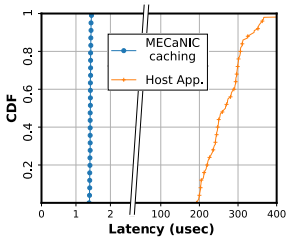
Fig. 11b shows the receiving rate passing through each VM, and we can check that *MECaNIC* successfully enforces its maximum bandwidth rate. At first, Flow A sends at a rate of 5 Gbps (0-5 sec), but the bandwidth for VM A is limited to 4 Gbps. When Flow B starts (5-10 sec), it is reserved 2 Gbps by taking part of the bandwidth allocated to Flow A, since their queue ranges ([2:9] and [2:5]) are distributed according to their criticality. Moreover, since the queue range [0:1] is reserved for Flow C, 1 Gbps out of the total 5 Gbps of bandwidth remains idle capacity that cannot be used by Flow A. It is only used after Flow C starts from 10 sec. Then, as the sending rate of Flow C increases, the receiving rates of VM 1 and 2 are adjusted by queue range and assigned by criticality.

B. Task offloading measurement

Pattern matching: *MECaNIC* can accelerate the performance of MEC applications by payload matching at the data plane level, allowing match results to be recognized as soon as applications receive packets. Here, we see how *MECaNIC*'s pattern match support can improve the performance of MEC

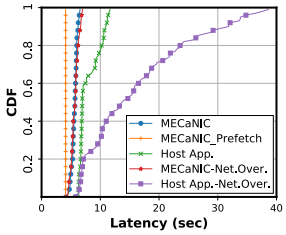


(a) TPS for packet transfer

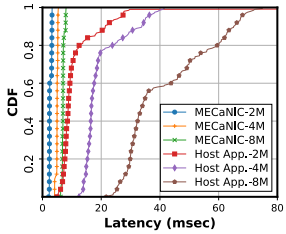


(b) Session establish time elapsed

Fig. 13: TCP session-packet transferring



(a) File download time



(b) Random access response time

Fig. 14: response caching benchmark

applications. The MEC VM runs Snort IDS [62] to perform payload search without other options and leverages this as an application that matches payload for several patterns. We then compare the throughput and latency for the cases when MEC applications handle packets by itself or not to determine match results based on the result received from *MECaNIC*.

Fig. 12 shows its result. Compared to simple forwarding (i.e., 0 patterns), the application causes significant performance degradation of 50% even when searching for a single pattern. Moreover, when processing over 100 patterns, the throughput is reduced by another 50%, which is only about 500 Mbps. The latency also increases significantly, especially as more patterns are processed. On the other hand, *MECaNIC* handles multiple patterns in parallel via hardware. Hence, the application does not incur a performance penalty from payload matching.

Response caching - TCP session: *MECaNIC* can manage TCP sessions and generate response messages at the data plane level by itself on the response caching. We evaluate how many transactions *MECaNIC* can handle in transferring data (i.e., transactions per second, TPS) by requesting arbitrary data of varying sizes (64-1514 bytes) from the packet generator.

As seen in Fig. 13a, *MECaNIC* can handle about 2M-185K sessions depending on packet transfer sizes. Also, since upcall to the host software layer is not needed, session establishment only takes 1-2 μs (Fig. 13b). Measuring these performances with a simple TCP session handler running on the host VM, the application can only manage 1.6K sessions and takes 200-400 μs to establish a session. This result indicates that *MECaNIC* can significantly improve the throughput of the MEC host and reduce its response time.

File download: To evaluate general response caching performance, we measure the time elapsed when downloading a 4 GB file stored in the MEC VM through *wget*. The *MECaNIC* workflow is shown in Fig. 6 in §III-D.

Fig. 14a shows its result, and we see that response caching significantly improves the file download response time. The download time that has taken around 8 sec on average and up to 12 sec through the naive wget command is reduced to 5.6 sec with *MECaNIC* response caching. Also, when the content is fetched into *MECaNIC* in advance like Fig. 7b, the time is further reduced and converges at 4 sec. Also, assuming network stress for proxy from a remote server, MEC VM is significantly affected and the average download time increases to 18 sec. However, since *MECaNIC* handles the file transfer at the data plane level directly without interference from/to the host, there is no difference in the download time.

Video streaming: To evaluate the improvement in response time through response caching, we implement a simple video streaming interface using Flask [63] on the host VM and measure the response time for seeking a random position in a video (i.e., random access) by the HTTP random range requests [64]. The time is measured from the time the search is requested to the completion of receiving 2, 4 and 8 MB data chunks corresponding to 480p, 720p, and 1080p resolutions, respectively. In the case of *MECaNIC*, it is assumed that the video is fetched in the task offloading module in advance, and it reads the request message by the pattern matching feature.

As seen in Fig. 14b, the response time for *MECaNIC* is very short for all chunk variations. Even for 8 MB (1080p), the maximum response time is less than 8 ms, which is equivalent to a seek of an arbitrary position in the video in less than one frame (16 ms) at 60 FPS by simple calculations, achieving near real-time user experience. In the case of the MEC VM, the average response times are 11.09, 19.95 and 41.39 ms respectively for each video chunk size, and even the fastest case is about 40% slower than the slowest case of *MECaNIC* (i.e., 1080p-8MB). Also, the response time reaches up to 75 ms, which is a perceivable delay to the end-user.

VI. Conclusion

The existing MEC architecture overlooks how the network should handle traffic for URLLC. Hence, packets for important services can be delayed or lost, and packet processing incurs considerable overhead in MEC hosts. To address this issue, this paper presents *MECaNIC*, a novel architecture that defines a hardware data plane for MEC hosts to provide precise scheduling and process offloading for URLLC. *MECaNIC* can effectively prioritize latency-sensitive traffic while ensuring reliable communication, even if packet contention occurs. Also, payload processing and response caching of *MECaNIC* reduce packet processing time in MEC applications and enable a faster response by mitigating host overhead. We believe that *MECaNIC* will play an important role in the Internet of Everything (IoE) era where more devices will be connected, from improving service quality to preventing fatal malfunctions.

Acknowledgement: We thank the anonymous reviewers and the shepherd for their careful reading of our manuscript and their many insightful comments and suggestions. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2022R1C1C1006967, 2021R1A4A1032252, and 2020R1F1A1076425) and Brain Korea 21 (BK21).

REFERENCES

[1] Report ITU-R M. 2410-0. Minimum requirements related to technical performance for imt-2020 radio interface (s). 2017.

[2] Haesik Kim. Ultra-reliable and low latency communication systems. 2020.

[3] Ning Lu, Nan Cheng, Ning Zhang, Xuemin Shen, and Jon W Mark. Connected vehicles: Solutions and challenges. *IEEE internet of things journal*, 1(4):289–299, 2014.

[4] Elisabeth Uhlemann. Introducing connected vehicles. *IEEE Vehicular Technology Magazine*, 10(1):23–31, 2015.

[5] Mohammad Aazam and Eui-Nam Huh. E-hamc: Leveraging fog computing for emergency alert service. In *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 518–523. IEEE, 2015.

[6] Melvin B Greer Jr and John W Ngo. Personal emergency preparedness plan (pepp) facebook app: Using cloud computing, mobile technology, and social networking services to decompress traditional channels of communication during emergencies and disasters. In *2012 IEEE Ninth International Conference on Services Computing*.

[7] Gautam S Thakur, Mukul Sharma, and Ahmed Helmy. Shield: Social sensing and help in emergency using mobile devices. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*.

[8] GSMECIEG ETSI. 004, mobile edge computing (mec) service scenarios v1. 1.1,(2015).

[9] Gerhard P Fettweis. The tactile internet: Applications and challenges. *IEEE Vehicular Technology Magazine*, 9(1):64–70, 2014.

[10] Martin Maier, Mahfuzulhoq Chowdhury, Bhaskar Prasad Rimal, and Dung Pham Van. The tactile internet: vision, recent progress, and open challenges. *IEEE Communications Magazine*, 54(5):138–145, 2016.

[11] Adnan Aijaz and Mahesh Sooriyabandara. The tactile internet for industries: A review. *Proceedings of the IEEE*, 107(2):414–435, 2018.

[12] Philipp Schulz, Maximilian Matthe, Henrik Klessig, Meryem Simsek, Gerhard Fettweis, Junaid Ansari, Shehzad Ali Ashraf, Bjoern Almeroth, Jens Voigt, Ines Riedel, et al. Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.

[13] Intel. Intel mobile edge computing technology improves the augmented reality experience. https://www.youtube.com/watch?v=ZSkWJYeKjnk.

[14] ISGMEC ETSI. Multi-access edge computing (mec); framework and reference architecture. Technical report, 2016, 2019.

[15] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[16] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[17] Angel Martin, Roberto Viola, Mikel Zorrilla, Julián Flórez, Pablo Angueira, and Jon Montalbán. Mec for fair, reliable and efficient media streaming in mobile networks. *IEEE Transactions on Broadcasting*, 66(2):264–278, 2019.

[18] Xiangwang Hou, Zhiyuan Ren, Kun Yang, Chen Chen, Hailin Zhang, and Yao Xiao. Iiot-mec: A novel mobile edge computing framework for 5g-enabled iiot. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–7. IEEE, 2019.

[19] Francesco Spinelli and Vincenzo Mancuso. Toward enabled industrial verticals in 5g: A survey on mec-based approaches to provisioning and flexibility. *IEEE Communications Surveys & Tutorials*, 23(1).

[20] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. Dfc: Accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 551–565, 2016.

[21] Taejune Park, Jaehyun Nam, Seung Ho Na, Jaewoong Chung, and Seungwon Shin. Reinhardt: Real-time reconfigurable hardware architecture for regular expression matching in dpi. In *Annual Computer Security Applications Conference*, pages 620–633, 2021.

[22] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, 2016.

[23] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, et al. Picnic: predictable virtualized nic. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[24] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[25] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. A case for smartnic-accelerated private communication. In *4th Asia-Pacific Workshop on Networking*, pages 30–35, 2020.

[26] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful tcp offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[27] NetFPGA. NetFPGA-SUME board. https://netfpga.org/site/#/systems/1netfpga-sume/details/.

[28] ETSI. Multi-access Edge Computing (MEC). https://www.etsi.org/technologies/multi-access-edge-computing.

[29] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[30] RedHat. Linux KVM: Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page.

[31] Taejune Park, Seungwon Shin, Insik Shin, and Kilho Lee. Formullar: An fpga-based network testing tool for flexible and precise measurement of ultra-low latency networking systems. *Computer Networks*, 185:107689.

[32] Kengo Sasaki, Naoya Suzuki, Satoshi Makido, and Akihiro Nakao. Vehicle control system coordinated between cloud and mobile edge computing. In *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 1122–1127. IEEE, 2016.

[33] Ahmed Nasrallah, Akhilesh S Thyagaturu, Ziyad Alharbi, Cuixiang Wang, Xing Shao, Martin Reisslein, and Hesham ElBakoury. Ultra-low latency (ull) networks: The ieee tsn and ietf detnet standards and related 5g ull research. *IEEE Communications Surveys & Tutorials*, 21(1):88–145, 2018.

[34] Hyoungju Ji, Sunho Park, Jeongho Yeo, Younsun Kim, Juho Lee, and Byonghyo Shim. Ultra-reliable and low-latency communications in 5g downlink: Physical layer aspects. *IEEE Wireless Communications*, 25(3):124–130, 2018.

[35] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[36] Vajiheh Farhadi, Fidan Mehmeti, Ting He, Tom La Porta, Hana Khamfroush, Shiqiang Wang, and Kevin S Chan. Service placement and request scheduling for data-intensive applications in edge clouds. In *2016 IEEE Conference on Computer Communications*, 2019.

[37] Sheng Yue, Ju Ren, Nan Qiao, Yongmin Zhang, Hongbo Jiang, Yaoxue Zhang, and Yuanyuan Yang. Todg: Distributed task offloading with delay guarantees for edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1650–1665, 2022.

[38] SAMSUNG. SmartThings Developers. https://developer-preview.smartthings.com/docs/getting-started/welcome.

[39] Philips. Philips Hue Developers. https://developers.meethue.com/develop/get-started-2/.

[40] AWS. AWS IoT Core: Connecting to AWS IoT Core service endpoints . https://docs.aws.amazon.com/iot/latest/developerguide/iot-connect-service.html.

[41] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[42] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[43] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[44] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *2020 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 243–259, 2020.

[45] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking

for the cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 681–693. ACM, 2020.

[46] Tianle Mai, Haipeng Yao, Song Guo, and Yunjie Liu. In-network computing powered mobile edge: Toward high performance industrial iot. *IEEE Network*, 35(1):289–295, 2021.

[47] Louiza Yala, Pantelis A Frangoudis, and Adlen Ksentini. Latency and availability driven vnf placement in a mec-nfv environment. In *2018 IEEE Global Communications Conference (GLOBECOM)*.

[48] Tianyu Yang, Yulin Hu, M Cenk Gursoy, Anke Schmeink, and Rudolf Mathar. Deep reinforcement learning based resource allocation in low latency edge computing networks. In *2018 15th international symposium on wireless communication systems (ISWCS)*, pages 1–5. IEEE, 2018.

[49] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.

[50] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[51] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019.

[52] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE access*, 4:5896–5907, 2016.

[53] Tiago Gama Rodrigues, Katsuya Suto, Hiroki Nishiyama, and Nei Kato. Hybrid method for minimizing service delay in edge cloud computing through vm migration and transmission power control. *IEEE Transactions on Computers*, 66(5):810–819, 2016.

[54] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11):1576–1589, 2018.

[55] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100, 2020.

[56] Ashiq Anjum, Tariq Abdullah, Muhammad Tariq, Yusuf Baltaci, and Nick Antonopoulos. Video stream analysis in clouds: An object detection and classification framework for high performance video analytics. *IEEE Transactions on Cloud Computing*, 2016.

[57] Gerald Sabin and Mohammad Rashti. Security offload using the smartnic, a programmable 10 gbps ethernet nic. In *2015 National Aerospace and Electronics Conference (NAECON)*, pages 273–276. IEEE, 2015.

[58] nVidia Mellanox. BLUEFIELD-2 DPU, 2020. https://resources.nvidia.com/en-us-vmware/bluefield-2-dpu-pb.

[59] Open vSwitch. An Open Virtual Switch. http://openvswitch.org/.

[60] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[61] NetFPGA-SUME. NetFPGA SUME Reference NIC, 2020. https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-NIC.

[62] Cisco. Snort - Network Intrusion Detection and Prevention System. https://www.snort.org/.

[63] Pallets. Python Flask. https://flask.palletsprojects.com/en/1.1.x/.

[64] R Fielding, Yves Lafon, and Julian Reschke. Hypertext transfer protocol (http/1.1): Range requests. *IETF RFC7233, June*, 2014.