On performance modeling for the management of Cloud-native Network Functions in closed-loops

Toni Dimitrovski[1, 2], Belma Turkovic[1], Aditya Ganesh[1], Timothy Lynar[3], Hans van den Berg[2], and Geert Heijenk[2]

[1] Networks Department, TNO
[2] Design and Analysis of Communication Systems, University of Twente
[3] School of Systems & Computing, University of New South Wales

***Abstract*—Allocating guaranteed resources and optimizing system parameters for Cloud-native Network Function (CNF) deployments introduce significant complexity in management. The performance of CNFs deployed on shared resources always depends on their incoming traffic and their competition for the underlying resource, both of which are usually very dynamic. The complexity of managing their deployments across many physical resources can explode very quickly and become infeasible for taking runtime management decisions in closed-loops. In this work we experimentally analyze the impact of CNFs sharing Central Processing Unit (CPU) and memory resources and make a key observation that increasing the number of deployed CNFs results in an increase of packet service time variability which in turn leads to longer waiting times. We also propose a model of CNF performance when sharing these resources, and discuss its extension to a general compute-network resource model for CNF performance estimation in closed management loops.**

***Index Terms*—CNF, Performance Model, UPF, CPU, memory, resources, Cloud Computing, Closed-Loop Management**

I. Introduction

With the recent advances in virtualization and the move of 5G/6G Network Functions (NFs) to cloud infrastructures, Cloud-native Network Functions (CNFs) that process user packets can be deployed as containers with shared resources. They can access the underlying resources with guarantees or through sharing mechanisms, in which case the host ensures that all functions get a fair share of that resource [1] [2].

With the advances of network automation, network design is embedded in the closed-loop orchestration of the computing infrastructure. Aside from considering links, this process also has to consider the CPUs [3] and memory, because these are shared among CNFs and possibly other compute workloads. One of the most important aspects is how this sharing impacts the performance of CNFs. This is especially relevant in resource constrained edge environments for In-Network Computing, where there is not much space for the resources to be partitioned, allocated and guaranteed. The partitioning also decreases efficiency gains from virtualization and increases the complexity of management due to the explosion in combinatorial options when optimizing performance [4], so infrastructure models with resource sharing impacts would be useful.

The decision when and how to deploy and configure these CNFs in the cloud infrastructure is made in the Management & Orchestration system [5], illustrated in Figure 1. Making this system intelligent by implementing algorithms that require

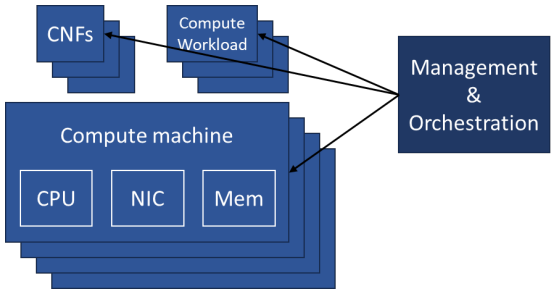

Fig. 1: Computing infrastructure, workloads and orchestration

feedback on the result of their actions is infeasible because performing actions on production infrastructure to find the optimal solution may decrease performance. To enable learning in practice, models that represent the infrastructure and its resources accurately are needed.

In this context, we take the first step towards developing a general compute-network resource model which facilitates autonomous design of networks, and focus on the following research question: "To what extent can a simple model accurately represent the sharing of CPU and memory resources and be also scalable for use in the runtime orchestration of CNF deployments on hundreds or thousands of machines?". Our contributions towards answering it are (i) experimentally showing that sharing the CPU and memory by multiple CNFs results in increased packet waiting times at the incoming Network Interface Card (NIC) as a consequence of increased packet service time variability dependent on the number of deployed CNFs, (ii) a method to find the parameters of a known Queueing Theory (QT) model that predicts the CNF performance in terms of packet latencies and (iii) express the first and second moment of the CPU packet service time distribution as a function of the number of deployed CNFs in the model. These contributions result in a general model capturing the CNF's sharing of CPU and memory resources. To discuss these aspects we use a network function called the User Plane Function (UPF), which is the packet processing function in the 5G core network.

The structure of this work is as follows. In Section II we provide related work on NF performance models and their use. In Section III we describe the system from the perspective of packet forwarding. Section IV presents details of our lab setup with an instance of the system and key observations

978-3-903176-75-1 ©2025 IFIP

from experiment measurements. In Section V we combine system knowledge with the observations to choose a model and estimate its parameters. Section VI evaluates the model performance and discusses its use. Finally, conclusions and future work are in Section VII.

II. Related work

Performance of computer and communication systems is often modeled using QT and Machine Learning (ML). For example, the authors in [6] assume a specific performance curve and fit it to the measurement data of different types of Virtual Network Functions (VNFs), generating profiles. However, the measurement setup used is a single compute machine with a single deployed VNF and the virtualization layer is out of scope. Hence, an assumption is taken that to use this profile a hard resource reservation must be done so the VNF has all resources it needs at all times. Building on the previous work, [7] uses the VNF profiles and develops a procedure to derive performance models of specific Service Function Chains (SFCs) that contain a few VNFs. This involves methods to fix the errors of using the VNF profiles which are a consequence of standalone measurement. The source of the errors comes from the competition by multiple VNFs on the same resource. The authors identify them as a consequence of "network Input/Output (I/O) bottlenecks in the underlying infrastructure, which is dynamically influenced if multiple VNFs are competing for the same network I/O".

The I/O bottlenecks referred to above have been discussed in literature, but often as future work. Most recently, the authors of [8] studied the tail latency of containerized packet processors, revealing that cache-sharing among cores can have a big impact on performance. However, the analysis was performed with only single instances of functions on a single machine, leaving the shared system resource for future work. This aspect has been discussed some time ago by [9], and more deeply analyzed in [10], where the authors identify and measure three reasons how the CPU's and the NIC's access to memory can lead to contention among NFs leading to performance degradation. They argue there are multiple sources of contention and all of them have different indicators but conclude that their sensitivity can be complex. This is also reflected in their methodology which produces accurate models, but it is not a scalable process considering the myriad of NFs, compute machines and optimization parameters.

In summary, the solutions in literature are either based on data that does not contain the effect of NFs sharing CPU and memory resources or they attempt to model the effect with many details, parameters and Key Performance Indicators (KPIs), all of which are not feasible to be processed in a production environment. Hence, a general model capturing the effect but is also scalable for runtime orchestration is missing.

III. System Description

Our system of interest is a single machine from a general cloud infrastructure. In this section we describe this system from the perspective of packet forwarding and its performance.

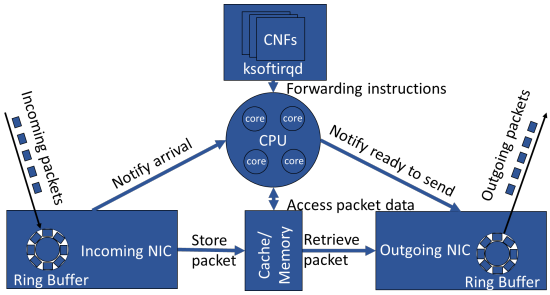

Fig. 2: Packet forwarding process

A. Packet forwarding

The packet forwarding process in a machine is illustrated in Figure 2 and starts with the ingress NIC. It typically has multiple transmit and receive queues, stored in local memory structures called Ring Buffers (RBs). The RBs are circular and in case of slow processing may lead to oldest packets being overwritten, which is how packet drops occur. The NIC uses a hashing function on the header of the packet in order to decide which queue to place the packet in. The information the NIC uses here is usually configurable but in most cases it is a 5-tuple of Internet Protocol (IP) source and destination addresses, transport layer protocol used and its ports. Since packets from the same flow need to be served in sequence, this ensures that they end up in the same queue.

After the output of the hashing results in a specific queue, the NIC will take the next available location and store the packet in cache or memory. Next, the NIC will trigger an interrupt, notifying the CPU that packet processing is needed. Each queue of the NIC interrupts and is served by a different core of the CPU. When the core is available it masks the interrupt for that queue and executes instructions to forward the packet which, among other operations, involve cache or memory accesses. In principle, the CPU follows a hierarchical approach and attempts to get data from the fastest L1 cache first, and the memory last. This may slightly vary depending on the architecture, but it always results in an increase of the packet service time when it happens. The length of the increase varies and is in the order of 1-2 cycles if the data is found in L1 cache, 5-10 cycles in case of L2, 10-50 cycles for L3 and 50-100 cycles or more in memory. The amount of cycles stack in case of misses, resulting in a sum of cycles spent for each previous level missed. The main reasons for cache misses are not enough capacity and conflicts with different data mapping to the same cache location. In multi-core systems the L1 cache is usually per core but L2 and L3 are usually shared.

The packet forwarding is performed in an Operating System (OS) kernel process, which in Linux is called *ksoftirqd* [11] and runs per core. After a configured number of packets (or all) are forwarded, the core stops executing *ksoftirqd* and gives time to other processes running on the machine resulting in a packet forwarding pause. If all of the packets in the queue were processed the core also unmasks the interrupt for that queue. In case the core does not have another process to execute and it served all the packets in the queue it may go into one of the

lower energy CPU Idle States, called *C-States*. If this is not the case it reschedules *ksoftirqd* to be executed after some time is given to other processes. The unmasking of the interrupt and going into a lower energy *C-States* cost more time to come back from relative to the rescheduling of *ksoftirqd* and can impact the waiting times of the first few incoming packets. This event handling mechanism in cycles is called NAPI [11].

B. Fast processing and multiple UPFs

Forwarding the packet in the section above means all the operations needed to determine the output NIC and notifying it to send the packet out. A recent OS development that gives much more flexibility with regards to packet processing is eXpress Data Path (XDP), resulting in a significant increase in packet processing speed. When one or multiple UPF containers are deployed on a machine, each with its own forwarding table, they get offloaded to the kernel space XDP program and executed upon arrival of packets. Depending on which UPF the packets belong to it uses the appropriate forwarding table. However, this is still done by the *ksoftirqd* and XDP program, so the addition of UPFs does not mean different packet forwarding processes.

C. CPU performance indicators

The CPU's overall performance is measured in the number of Instructions Per Second (IPS) it can execute. However, modern CPUs execute instructions in parallel within a pipeline, resulting in improved efficiency but prone to execution stalls in the case of inter-dependencies between the instructions. We are interested in this due to the introduction of multiple UPFs populating our forwarding tables, which may introduce more memory-related operations. Thus, it is also important to look at another indicator called Cycles Per Instruction (CPI) which gives the average number of cycles used per instruction. It shows the CPU efficiency which decreases when more cycles are spent on the same instructions.

IV. System experiments

In order to analyse the impact UPFs have on each other when sharing resources without reservations or optimizations, we set up lab experiments with UPFs running on a machine. In this section we present this setup and make key observations from measurements done in the performed experiments.

A. Lab setup

Due to ease of management, we used a Virtual Machine (VM) hosted on an a SuperMicro COTS x86-64 server of a private OpenStack cloud to deploy containerized Generic-XDP accelerated Open5GS[1] UPFs. The VM was provided with HugePages, pinned to 16 cores of an Intel Xeon Gold 6130 CPU. CPU frequency scaling was disabled and all cores of the processor were fixed to a frequency of 1GHz in order to avoid confounding effects and make the CPU a bottleneck. The VM was attached with Single Root Input/Output Virtualization (SR-IOV) to two *Mellanox* Technologies MT27710 NICs, each with a bandwidth of 10 Gbps. These features, together with the fact no other process ran on the physical machine, kept the hypervisor overhead negligible. Each NIC input queue was set to max length of 1024 and fixed to an individual CPU core. The machine and VM run Linux OS kernel version 5.4. The VM was connected through the NICs to a physical server running the T-Rex traffic generator that sent packets via the UPFs and received them back. Each measurement consists of 60 seconds of traffic at a given rate, split equally across a predetermined number of UPFs and user flows. Measurements were taken for 1, 5, 10, 20 and 25 UPFs at packet rates ranging from 3 Kpps to 3 Mpps which is the theoretical limit of the NICs for the chosen packet size of 700B. Important to note is that the inter-packet intervals are constant because T-Rex does not support random intervals. Loss and delay metrics were obtained from observing the round-trip packet behaviour (called response time in the following sections) at the T-Rex machine. Each configuration was iterated 50 times. The capacity C of the machine set up in this way is 1.82 Mpps, which is the maximum load that the system was able to reach in all experiment iterations.

[1]https://github.com/open5gs/open5gs

B. Observations

From the experiments performed with the settings described above, we can make the following observations:

O.1 The increase of load generally means more instructions to execute per unit time so the CPU can better utilize the pipeline execution capabilities leading to higher efficiency in packet forwarding and lower CPI. However, the CPI also increases with the increase of the number of deployed UPFs partly diminishing this effect and decreasing the efficiency gain, as observed in Figure 3. With the increase of the number of deployed UPFs the CPU needs more cycles to perform the same number of instructions. The underlying reason for this is that more cycles are spent on memory access because operations that perform packet forwarding need alternating data from different UPFs. Since packet forwarding instructions are inter-dependent, the cache misses lead to execution stalls. This can be seen in Figure 4, which shows the number of execution stalls (perf event *cycle_activity.stalls_l*_miss*) when the CPU is waiting for data due to a cache miss. Different levels of cache are represented by different line style. The number of deployed UPFs clearly increases the number of stalls and this effect grows at higher loads. The values shown in the figure are an average for a given configuration, so their distribution is unknown.

O.2 Figure 5 shows the mean, median and 99.9th percentile of the packet response time distribution dependent on the number of UPFs. Each plot shows these three values for three different traffic loads - Low=17%, Mid=51% and High=88%. It is visible that the response times experienced on the machine have long tails even for a single UPF, as identified by many previous works in literature [8] [12]. In low loads this can be attributed to the interrupt handling procedure and CPU *C-states*, described in Section III, because it takes more time for the CPU to start forwarding packets when they arrive in the

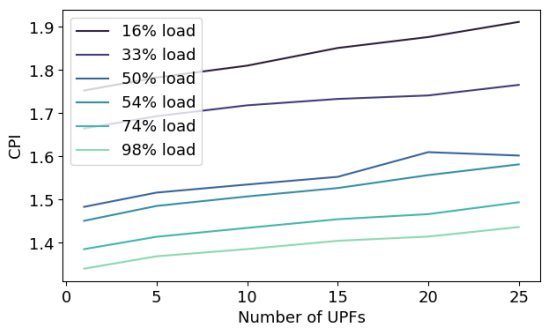

Fig. 3: Cycles per Instruction

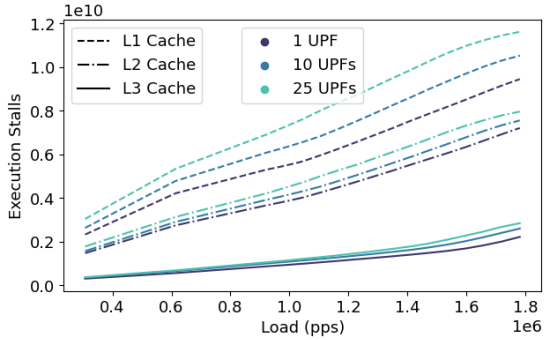

Fig. 4: Execution stalls

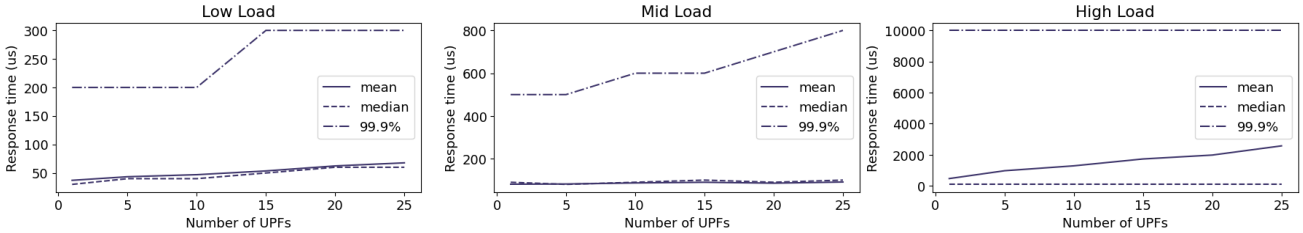

Fig. 5: Mean, median and 99.9th percentile of all measurements with different number of UPFs

queue. In mid to high loads when this almost never happens it can be attributed to the pauses from packet forwarding which have a similar effect. Although one can expect lower intensity of the effect here because rescheduling is faster than interrupt handling, the opposite is true because the traffic load is higher which results in more packets being queued. The effect of the pauses is amplified when the CPU is shared between UPFs and its impact increases with the increase of their number. This is the result of the stalling discussed in the previous observation. In low loads when the interrupt handling is prevalent its effect is a complete move of the response time distribution to the right. All three indicators increase with the increase of deployed UPFs. This means it is more equally spread over all of the packets. However, in mid to high loads, the median is stable around the same value but the mean and 99.9th percentile grow which indicates that the stalls in combination with higher loads generate more heavy events with long queues and high response times.

V. Performance Modeling

Having described the system and given some observations, in this section we introduce the system model and a method to estimate the model parameters. Since our focus is on CPU and memory, we exclude the outgoing NIC which is why we made the CPU a bottleneck as described in section IV-A. Furthermore, we assume the hashing function is ideal and equally spreads the load across cores. In the experiment runs this is achieved by sending a sufficient number of different header tuples resulting in core load differences below 20%.

A. System model

The fact that each queue has its own core makes that each core behaves like a single independent server with the same performance on average. Hence, we consider a system model (illustrated in Figure 6) with P servers representing CPU cores, each with its own queue in the incoming NIC and $P\lambda$ total incoming average packet rate. The NAPI cycle behavior with packet serving pauses makes the CPU core behave as an M/G/1 cyclic service system with exhaustive k-limited service strategy and switchover times studied in [13] where an approximation of the expected waiting time $E[W]$ is given. However, they consider a system with multiple input queues per server whereas we have a single queue per server. It is also important to note that the switchover times from this model represent both types of packet forwarding pause - the case when a core unmasks the interrupt and the case when it reschedules *ksoftirqd*. Depending on the difference between the pause length distributions in the two cases, this simplification may introduce inaccuracy.

Let β be the mean and $\beta^{(2)}$ the second moment of the packet service time distribution, which represents the time the core spends to forward a packet. The switchover time represents the time the core is on a pause from packet processing with mean s and second moment $s^{(2)}$. Since we only have a single queue, k is a configuration parameter representing the number of packets served in a NAPI cycle, λ is the input packet rate per queue and $\rho = \lambda\beta$. The approximation of the expected waiting time in the system from [13] then becomes:

$$E[W] \approx \frac{(1-\rho)^2 + \frac{\rho}{k}(2-\rho)}{1-\rho-\frac{\lambda s}{k}} \frac{\rho\frac{\lambda\beta^{(2)}}{2(1-\rho)} + \rho\frac{s^{(2)}}{2s} + \frac{\rho^2 s}{k(1-\rho)}}{[\rho(1-\rho) + \frac{\rho^2(2-\rho)}{k(1-\rho)}]}. \quad (1)$$

B. Method for estimating model parameters

Since we have a dataset that contains measurements of response times under different number N of deployed UPFs for different input packet rates λ, we can use it to fit the model

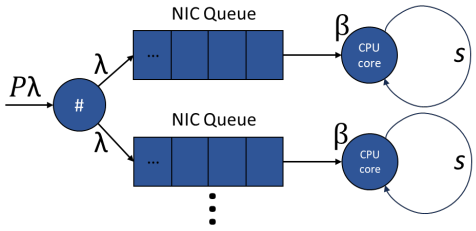

Fig. 6: System model

and estimate its parameters $\beta, \beta^{(2)}, s, s^{(2)}$. Measuring these require system modifications which are infeasible at runtime so they need to be derived. Assuming that the packet service time distributions will be different for different N due to observation O.1 from Section IV-B we perform the following steps, further detailed below: (i) set the model capacity to the data point at maximum system capacity C under one deployed UPF; (ii) fit the model to the rest of the data points with one deployed UPF and estimate the packet service time and switchover time distribution parameters; (iii) use the estimated switchover time distribution parameters for one deployed UPF and the data points with multiple deployed UPFs to estimate the different packet service time distribution parameters for each number of deployed UPFs; (iv) establish the relationship between the number of deployed UPFs and packet service time distribution parameters, and substitute it in the original model.

(i) Assuming system capacity C in packets per second, we experimentally determine the packet rate λ_∞ at which we reach C per core by $\lambda_\infty = \frac{C}{P}$. In our system model from Equation (1) this happens when the first term reaches infinity, which is when its denominator $1 - \lambda_\infty \beta - \frac{\lambda_\infty s}{k} = 0$. Since we now know λ_∞, we can express the mean switchover time s in terms of the mean service time β:

$$s = \frac{k(1 - \lambda_\infty \beta)}{\lambda_\infty} \tag{2}$$

We then substitute s back into Equation (1), the result of which is setting the asymptote of $E[W]$ to λ_∞.

(ii) In the second step we use $f_\pi = E[W] + \beta$, which is the packet response time, in the objective for the Damped Least-Squares (DLS) [14] algorithm and fit the model to the rest of the data points with $N = 1$ to estimate our parameters $\pi = (\beta_{N=1}, \beta^{(2)}_{N=1}, s^{(2)}_{N=1})$. In order to minimize the relative instead of the absolute error, we take the natural logarithm from the response times, resulting in the objective

$$min_\pi \sum_{i=1}^{n} (ln \frac{y_i}{f(x_i, \pi)})^2 \tag{3}$$

where we have n number of samples with x_i input system loads, y_i measured response times and π function parameters.

(iii) As discussed in Section III-B, the addition of UPFs does not create multiple packet forwarding processes so it will only negligibly impact the switchover time because the user space processes do not need any processing resources. Hence, for the cases where $N > 1$, we can take the same switchover time mean and second moment $s^{(2)}$ as estimated for $N = 1$ and only use DLS with $\pi = (\beta_N, \beta^{(2)}_N)$. We iterate this for the measurement data obtained for all N.

(iv) The previous step results in parameter estimations $(\beta_N, \beta^{(2)}_N)$ for each value of N in the data. We then express β and $\beta^{(2)}$ as a function of N and substitute this in Equation (1).

VI. Results and Evaluation

In this section we present and evaluate results of the parameter estimation for the model using the method described in Section V and data from experiments in Section IV.

A. Model parameter estimation

Taking the system capacity C from Section IV-A, in Step(i) from the method in Section V-B we calculate $\lambda_\infty = \frac{C}{P} = 113782pps$ and substitute s in $E[W]$. Then using the updated $E[W]$ and solving the optimization problem from Step (ii) with DLS yields parameters $\beta = 8.74\mu s, \beta^{(2)} = 1.12ns^2, s = 13.52\mu s, s^{(2)} = 0.3ns^2$.

The results of Step (iii) are depicted on Figure 7, which shows β on the left *y-axis* and $\beta^{(2)}$ on the right *y-axis* for different N on the *x-axis*. It can be observed that with the increase of N the parameter β remains the same but the parameter $\beta^{(2)}$ increases in the depicted interval of N. This is a consequence of the execution stalls seen in Section IV-B O.1, which lead to higher variability of packet service time, which in turn leads to longer queues.

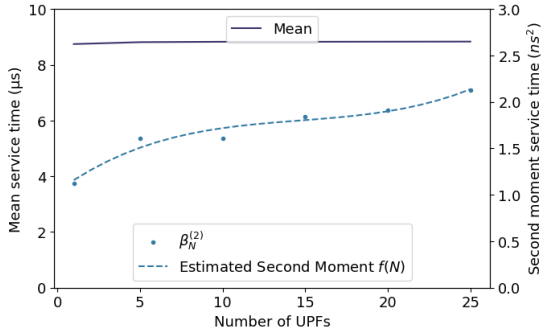

Fig. 7: Mean and second moment of packet service time

Using the estimated values of $\beta^{(2)}_N$, we can now express $\beta^{(2)}$ on this machine as a function of N. We estimate the coefficients with polynomial regression resulting in a cubic polynomial $\beta^{(2)} = \frac{1.78}{10^{13}} N^3 - \frac{7.84}{10^{12}} N^2 + \frac{1.28}{10^{10}} N + \frac{1.03}{10^9}$ which is the minimal degree that captures the three different rates of increase - between $N = (1, 5), N = (5, 20)$ and $N = (20, 25)$. This is depicted with a dashed line in Figure 7. We then substitute $\beta^{(2)}$ back into the original approximation of $E[W]$ together with $\rho = \lambda \beta_{N=1}$, which is the final adaptation of the model to support the prediction of performance depending on the number of deployed UPFs.

B. Modeling results and accuracy

The estimated mean packet response time by the model overlayed with the measurements for $N = 1$ are shown in Figure 8, and for all $N = \{1, 10, 25\}$ in Figure 9. A zoom of the *y-axis* in the 20-70% range of the *x-axis* is embedded in both figures to improve visibility. One can observe that the

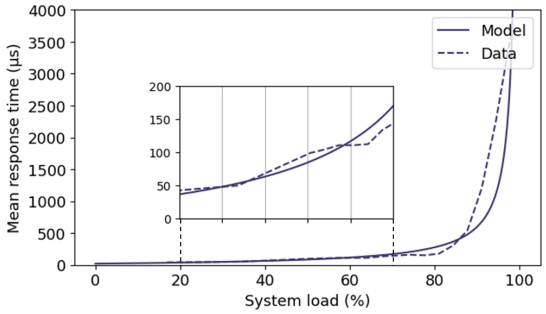

Fig. 8: Model results for N=1

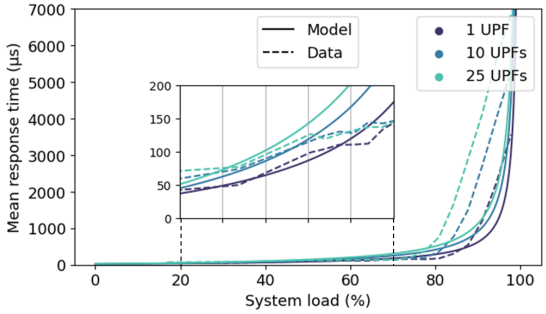

Fig. 9: Model results for N=1,10,25

mean response time gradually increases with system load and the rate of increase grows with the number of deployed UPFs. In fact, what happens is the response time distribution has more heavy events with long queues as observed in O.2.

The model accuracy between 60% and 99% load decreases due to the limitation in the testbed where the inter-packet time intervals of the input traffic are constant. Thus, we actually have a cyclic service system with exhaustive k-limited service strategy and *deterministic* arrivals instead of Poisson arrivals as in the approximation in our model. This causes the system to unmask the interrupt more often than with random arrivals, which we observed in the interrupt counts. The effect increases the mean waiting times in low and mid loads, but is only dependent on the load and is equally present for all N. It does affect the parameter estimation though, with the Mean Relative Error (MRE) for $N = 1$ being 0.24. In principle, when the input traffic gets more randomized or Poisson-like the error is expected to decrease. The distribution and magnitude of the errors are the same for $N > 1$.

VII. Conclusions and Future Work

With this work we have experimentally shown that when multiple UPFs share CPU and memory resources on a general purpose compute machine, they impact each other by increasing the variability of packet processing times. This leads to longer queues on the input NIC and a performance degradation, which needs to be managed in order to achieve Service Level Agreements (SLAs). We have also developed a method to fit a k-limited service model from QT to data from a compute machine. Furthermore, we adapted the model to be able to predict performance dependent on the number of UPFs deployed on the machine. One aspect for improvement of the model presented in Section V-A is modeling the interrupt and *C-states* handling with a separate parameter, using current knowledge from QT with setup times.

In order to have a complete compute-network model we need to include the NIC, resulting in a Digital Twin of the machine. Closed-loop management using the twin does not need actions to optimize underlying parameters but only decide placements if UPFs fit on the machine considering the performance. Resources can be used closer to their limits and unused machines switched off, perhaps obsoleting power saving measures like CPU frequency scaling. The method can also be used to derive parameters for different hardware. Its scalability stems from the fact it only needs the system load, number of UPFs and measured response times rather than extensive data collection [9] [10].

Acknowledgments

This work was partially funded by the EC through the SNS JU Hexa-X-II project under agreement No.101095759 and the Dutch National Growth Fund through the FNS program. The authors are also thankful to Suzan Bayhan, Alejandro Calvillo-Fernandez, Constantine Ayimba, Borgert van der Kluit and Milan Groshev for the fruitful discussions on the topic.

References

[1] S. Susnjara and I. Smalley, "What is containerization?" 2023. [Online]. Available: https://www.ibm.com/think/topics/containerization

[2] M. Jayakumar, "Why use containers and cloud-native functions anyway?" 2021. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/containers-and-cloud-native-functions-white-paper.pdf

[3] K. Pandit *et al.*, "Modeling the impact of CPU properties to optimize and predict packet-processing performance," 2018. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/case-studies/att-cpu-impact-on-packet-processing-perfomance-paper.pdf

[4] P. Liu and J. Guitart, "Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study," *The Journal of Supercomputing*, 2020.

[5] W. Attaoui *et al.*, "VNF and CNF placement in 5G: Recent advances and future trends," *IEEE Transactions on Network and Service Management*, 2023.

[6] S. Van Rossem *et al.*, "Profile-based resource allocation for virtualized network functions," *IEEE Transactions on Network and Service Management*, 2019.

[7] S. Van Rossem, "VNF performance modelling: From stand-alone to chained topologies," *Computer Networks*, 2020.

[8] F. Wiedner *et al.*, "Performance evaluation of containers for low-latency packet processing in virtualized network environments," *Performance Evaluation*, 2024.

[9] M. Dobrescu *et al.*, "Toward predictable performance in software packet-processing platforms," in *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, p. 11.

[10] A. Manousis *et al.*, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of SIGCOMM '20*, 2020, pp. 270–282.

[11] "NAPI," 2022. [Online]. Available: https://wiki.linuxfoundation.org/networking/napi

[12] S. Gallenmüller *et al.*, "Ducked tails: Trimming the tail latency of(f) packet processing systems," in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021.

[13] S. W. Fuhrmann and Y. Wang, "Analysis of cyclic service systems with limited service: Bounds and approximations," *Performance Evaluation*, 1988.

[14] D. W. Marquardt, "An algorithm for least-squares estimation of non-linear parameters," *Journal of the Society for Industrial and Applied Mathematics*, 1963.