Towards Reducing Last-Level-Cache Interference of Co-located Virtual Network Functions

Raphael Durner, Christian Sieber, Wolfgang Kellerer
Chair of Communication Networks
Department of Electrical and Computer Engineering
Technical University of Munich, Germany
Email: {r.durner, c.sieber, wolfgang.kellerer}@tum.de

***Abstract*—Network Function Virtualization (NFV) aims to virtualize compute resources for packet processing in order to gain flexibility and reduce costs. In order to increase the resource utilization, multiple VNFs are co-located on one single server. Current virtualization techniques do not fully isolate all resources, thus co-location of VNFs causes interference effects. It has been shown that these interference effects can degrade the performance of Virtualized Network Functions (VNFs) in terms of throughput and delay severely. In this work we aim to gather the potential that lies in reduction of the interference due to the shared Last Level Cache (LLC). CPU caches are used to improve the access times to memory that is needed regularly for the execution of a program. Intel Cache Allocation Technology (CAT) provides the means to allocate the cache and isolate VNFs from each other. The results show that the scheduler can decrease the CPU utilization by up to 20%. We can show which factors influence the gain of LLC scheduling in NFV deployments. In order to show this we propose a scheduler which optimally allocates the LLC in order to reduce the maximum CPU utilization of all cores.**

***Index Terms*—Network Function Virtualization; Processing-Resource Sharing; Memory Cache**

I. Introduction

With the move towards Network Function Virtualization (NFV), network providers move away from dedicated hardware appliances towards software running on commodity hardware. The commodity hardware is consolidated in a resource pool following the example of common compute cloud systems. Virtual Network Functions (VNFs) are deployed on this resource pool as needed by the network conditions. This abstract resource pool consists of physical compute servers, network switches and storage servers. On the one hand, this design pattern enables an improved usage of the available resources, leveraging the multiplexing gain. On the other hand, this pattern causes interference between the consolidated VNFs at different places in the shared system. [1] shows that VNFs are competing for network IO bandwidth, CPU, cache and memory. As a result, co-location can half the throughput of one VNF compared to the case the when VNF is running alone due to interference.

In this work, we concentrate on one specific interference effect caused by the co-location of VNFs on one single CPU chip: the Last-Level-Cache interference. In modern multi-core processors, some of the on-chip resources such as the Last-Level-Cache (LLC) are shared between all cores, which causes interference. To resolve this issue some chip manufacturers like Intel [2] or Qualcomm [3] are providing means to explicitly allocate shares of the LLC to specific cores and processes.

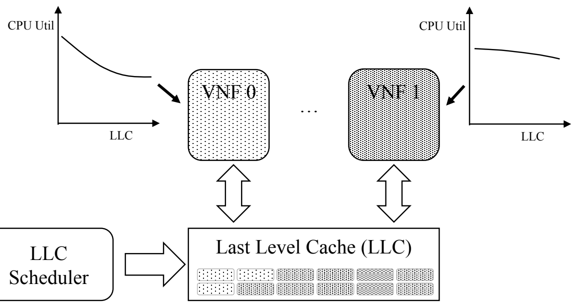

Figure 1. Scenario: Co-located VNFs share the LLC of one chip. This causes interference. As the performance gain of VNFs using a certain amount of LLC differs, the LLC allocation can be optimized using a scheduler.

Figure 1 shows the overall scenario. Multiple VNFs are running on the same CPU sharing the LLC. The caches of the CPU are fast on-chip memories and are employed to provide data that is used by programs regularly. The advantage is that the time necessary for accessing data in the caches is much lower than in the main memory. As a result the arithmetic units of the CPU have to wait less for data and thus the utility of the CPU is increased. If we consider VNFs that are not fully loaded this increase in utility reduces the CPU utilization. As VNFs can realize diverse functionalities (e.g. IDS, NAT, Firewall, VPN Gateway, billing) the influence of the size of the LLC is also diverse. VNF 0 in Figure 1 profits a lot from LLC while VNF 1 does not profit much. The LLC Scheduler tries to optimize the LLC allocation in order to decrease the CPU utilization. Our scheduler does not aim to decrease the utilization of single cores but instead optimizes the CPU utilization of the complete server.

In order to study the problem in depth, we model memory access patterns of VNFs and develop an emulator that enables us to study the problem without disturbance by other interference effects. In this work, we focus on static VNFs with memory access patterns that do not change over time.

One important question is how much gain can be expected in NFV deployments from LLC scheduling. We show that the maximum CPU utilization of all cores can be reduced by up to 20%, especially if the co-located VNFs are diverse in terms of load. Furthermore the working set size of the VNFs must be large enough to achieve gains.

Existing approaches such as ResQ [4] rely on detailed pre-built VNF profiles to schedule the LLC. Besides the increase in complexity, profiling of the VNFs can be difficult due to the dynamic nature of the network traffic that has a large impact on the VNF performance. To the best of our knowledge, there exists no other algorithm that is scheduling the LLC without any prior information of the VNFs in the literature so far. Thus, the main contributions of this work are as follows:

I) Modeling of the memory access patterns of VNFs and design of an emulation
II) Design of an optimal LLC scheduler for static VNFs
III) Evaluation of system parameters that influence the gain of LLC scheduling in NFV deployments

The paper is structured as follows: We first show related work in Section II. In Section III we describe the system architecture of the investigated server hardware in detail and describe the role of LLC-scheduling in NFV orchestration. Then we model the access behavior of VNFs to memory and it is explained how we emulate this access behavior (Section IV). In Section V we describe the scheduler algorithm and discuss its optimality. We show the inert behavior of the LLC and identify important metrics that influence the achievable gain by LLC scheduling in Section VI. Finally we conclude in Section VII.

II. Related Work

LLC interference in NFV environments was not widely studied yet, nevertheless there are some works which study LLC interference and aim to reduce LLC interference [1], [5], [6], [7]. NFV also emerged from cloud compute concepts, but LLC interference scheduling is studied more in depth with respect to compute cloud environments.

A. LLC Interference in Compute Cloud Environments

A number of works considers LLC contention effects in compute cloud environments [8], [9], [10], [11], [12], [13], i.e. not considering NFV. One main difference is the performance metric employed: this is commonly the completion time of a program. VNFs are different in nature as they are event based, an incoming packet is an event that has to be processed. Consequently a VNF program never completes its task and no completion time metric exists. Though some works can also give indications to the problem we are addressing here and were helpful during our work.

[9] studies performance degradation due to LLC interference. It is shown that programs that have many LLC references degrade stronger while other programs that are more compute intensive are less affected. We cover this diversity as we emulate VNFs with a larger and a smaller working set and show that the same findings are also true in NFV environments.

Heracles [12] reduces contention between batch tasks and event based tasks in order to improve the utilization of shared server resources. Heracles considers different resources such as network bandwidth, memory bandwidth CPU cores and also the LLC. The approach uses Intel CAT in order to isolate between the two types of tasks but does not consider interference between tasks of the same type. In contrast to that we study contention between multiple latency sensitive tasks, in our case VNFs.

Another work [8] aims to build a fair LLC scheduler. Fairness is defined such that the performance degradation due to LLC interference is equal for all programs. Authors show that the degradation is higher for some programs than for others and that the developed scheduler can avoid this effect. Compared to our work there are two main differences: On the one hand a different optimization objective is chosen and, on the other hand, authors focus on compute workloads rather than VNFs.

PACMan [13] places VMs on different servers such that the interference between the programs is reduced. The approach first profiles the VMs and then consolidates the VMs on different servers. The approach doesn't consider a controllable LLC like we do and does not optimize online, but uses the VM's profile to optimize the VM placement. As the profile of VNFs strongly depends on the traffic profiling is more difficult in an NFV environment. Nevertheless VNF profiling an placement/LLC optimization would be an interesting extension to our work, which we also consider to study more in deep in the future.

B. LLC Interference in NFV Environments

Recently authors of [1] studied the interference effects of co-located VNFs in depth. They consider contention of network I/O bandwidth, CPU, memory and cache. With there measurements they can show that different types of VNFs cause different effects. A VNF that only read packets but does not modify them, like a gateway, has a different pattern than a VNF that modifies the packet, e.g. a load balancer. This is in line with our results that show that the cache interference effects depend on the memory access patterns.

A different possibility than CAT that prevents cache interference is page coloring. Authors of [5] study cache coloring to avoid cache interference in an NFV environment. Page coloring enables the separation of the cache in different cache regions as pages with different colors do not contend for the same cache ways. The authors cluster different memory buffer pages to different colors. Different clustering methods are applied: Element-based clustering, groups functions of the same type together, while flow based buffering uses the same page colors for groups of flows. We are studying a different approach that does not cluster but schedules individual VNFs independently and without any knowledge about the VNF implementation.

Veitch et. al [6] showed that Intel CAT can improve the performance of VNFs when a noisy neighbor is present. A noisy neighbor is a VNF or program that evicts cache

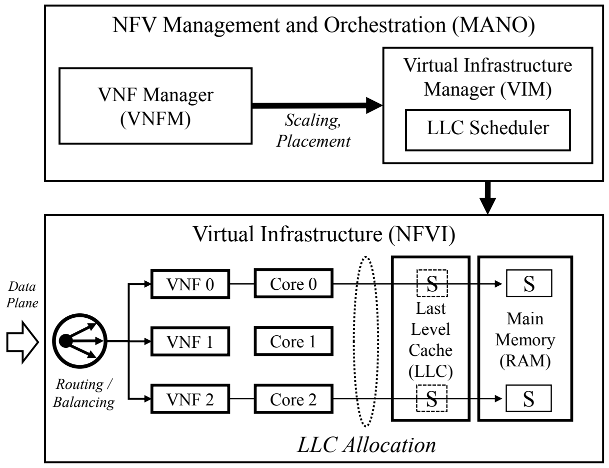

Figure 2. LLC scheduling in the bigger picture of NFV Management and Orchestration (MANO). Horizontal scaling of VNF instances for a specific service, e.g., a firewall, is performed by the VNF manager through the VIM. Through the routing/load-balancing functionality of the network, the VNFM dictates which fraction of the traffic is assigned to a specific VNF. Network elements then forward and distribute traffic to the running VNFs. The VNFs have to share the available LLC and only part of the working set *S* of each VNF can be kept in the LLC.

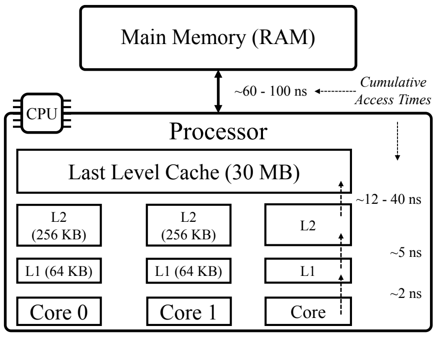

Figure 3. Simplified depiction of the Intel Xeon processor's cache hierarchy. Each core is equipped with an exclusive L1 cache (64 KB) and L2 cache (256 KB). A Last Level Cache (LLC) with a capacity of 30 MByte is shared. Access times are cumulative and range from roughly 2 ns if the memory access can be satisfied from the L1 cache to 100 ns if the data has to be fetched from the main memory.

lines regularly and therefore causes high interference and performance degradation to other VNFs co-located on the CPU. The authors study different static CAT configurations and show that the latency of VNFs can be decreased with CAT. In contrast to our work they aim to show the benefits that CAT can give in an NFV environment rather than aiming for an optimal allocation. Nevertheless this work was an important starting point for our work.

The work of Dobrescu et. al [7] studies interference effects of software packet processing systems (i.e. VNFs). The authors show that there exist different types of VNFs that use the cache in different manners. Some VNF types have only a low amount of LLC accesses while others have a high number of accesses. It is shown that these types impact the performance degradation when they are co-located on one CPU. This strongly supports our findings from Section IV. On the other hand the authors do not study LLC scheduling, but suggest an orchestration that places the VNF on different servers CPUs such that the interference is reduced.

ResQ [4] is the work that is closest to our work. It proposes to use CAT in order to enforce performance SLOs. The authors show that CAT can be successfully used to enforce throughput and latency guarantees. In contrast to our approach authors use an 2-step offline approach with pre-profiled network functions while we take an online approach without any a priori knowledge. First the network functions have to be profiled using a variety of traffic profiles. In the second step the profiles can be used to improve the placement of the NFs and the allocation of the LLC.

III. System Architecture

The section at hand first describes the role of LLC scheduling in the bigger picture of NFV. Afterwards, the cache hierarchy of the investigated NFV platform is introduced. The section is concluded with a brief description of the Intel Cache Allocation Technology and the cache monitoring mechanisms provided by Intel.

A. NFV MANO

Figure 2 depicts the LLC scheduler in the bigger picture of NFV Management and Orchestration (MANO). The architecture has been proposed by the ETSI NFV working group [14]. The architecture describes the components required in all stages of the life-cycle of a VNF, from the definition in terms of deployment and operational requirements, to the allocation and the release of the required resources. The proposed LLC scheduler can be implemented as part of the Virtualized Infrastructure Manager (VIM), the MANO component which manages the available resources of the physical infrastructure. With the presented scheduler, the VIM can optimize the distribution of the LLC to the active VNFs on the physical server. The amount of memory that is required from the VNF to fulfill its functionality is called working set. The working set consists of the binary and as well the state of the VNF such as tables, rules etc.

Commonly, a CPU core is assigned exclusively to a VNF to benefit most from processor register, L1 and L2 caching [15]. A load balancer on the data-plane, either in the network or in software on the host, distributes the network packets to the available instances. The goal of such packet load balancers is to keep the utilization homogeneous between the instances. Down and up-scaling of VNFs is done by the VNF Manager (VNFM) by stopping or starting additional VNF instances [16] in order to improve resource utilization. For example the VNFM may add a new instance at an average utilization of 90 % for all instances and remove an instance if the average utilization of all cores drops below 60 %.

B. CPU/Cache Hierarchy

The experimental set-up consists of a *Dell PowerEdge R530* server with 2 *Intel Xeon E5-2650 v4 2.2GHz* CPUs with 12 physical CPU cores each and the *Intel C610* chipset. We refer to CPU as one chip consisting of the cores and the LLC. Figure 3 illustrates the cache hierarchy of the Intel Xeon processor and the connection to the RAM. Each physical core is equipped with two exclusive cache levels, an L1 cache of 64 KByte and an L2 cache of 256 KByte. A Last-Level-Cache (LLC) of 30 MByte is shared among the 12 physical cores of the CPU. We disabled hyper threading on our server to eliminate this source of interference. The LLC is connected to the main memory, which consists of DDR4 RAM with a size of 32 GB.

Data from the main memory is accessed in chunks, denoted as *cache lines*, of 64 Bytes. When a CPU core accesses a particular memory location, the caches are checked incrementally starting from L1, through the shared LLC and up to the main memory. The caches of our CPU are inclusive, i.e., every line that is cached in L1 is also cached in L2 and LLC. The shared LLC is 20-way associative, hence data of every memory location can be cached at 20 locations in the LLC cache and each cache way has a capacity of 1536 KBytes.

C. Cache Allocation Technology

The Intel Cache Allocation Technology (CAT) [2] enables the allocation of the LLC to specific CPU cores. The allocation can be done shared, i.e., multiple cores share specific parts of the cache, or exclusively, i.e., parts of the cache are allocated to specific cores. In detail, CAT introduces 16 Classes of Service (CoS) for CPU cores. Each core has to be assigned exactly to one class, but multiple cores can be assigned to the same class. A bitmask per class configures which of the available 20 cache ways can be used by which CoS. In a nutshell, CAT enables the allocation of 20 LLC chunks, with each chunk having a size of 1536 KByte, to specific CPU cores. Due to limitations of the technology CAT only restricts write accesses to the LLC. This means that a core that had access to a larger share of the cache before a reallocation can still access cached data stored in ways that are allocated to a different CoS. This restriction causes an transient behavior of the cache after CAT changes. A detailed evaluation of the transient phase of the LLC is done in Section VI-B.

D. Monitoring

Intel processors provide low-level cache statistics via performance counters. These low-level counters can be read and interpreted using the Processor Counter Monitor (PCM) tool [17]. The developed algorithm uses the following metrics provided per core: current CPU utilization and LLC occupation.

IV. Memory Access Model

The sensitivity of VNFs to LLC contention, and also the interference caused by a VNF, depends on a number of factors, such as the size of the accessed memory range and also how often each memory location is accessed. In this section, we introduce a memory model that serves us for the derivation of the scheduler algorithm and which led us in the design of the memory access emulator.

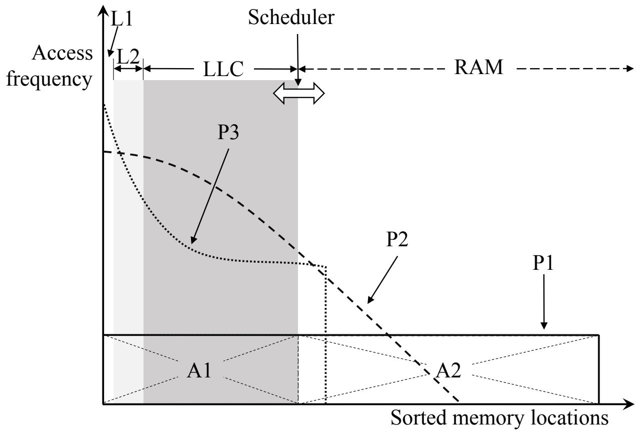

Figure 4. Possible memory access patterns.

A VNF, or any program, accesses different data with different frequencies, e.g., parts of the binary are accessed often while some other data might only be accessed for startup. Theoretically, we split the complete allocated memory of a VNF in chunks, e.g., each chunk is 64 Bytes large as in the cache lines. With this abstraction we can assign every chunk of memory a distinct access frequency and we can sort the chunks in decreasing order by the access frequency. In this work, we denote this as *access pattern*.

Figure 4 abstractly visualizes three different possible access patterns (P1, P2, P3). The horizontal axis represents the different memory chunks and the vertical axis of the plot represent the corresponding access frequency. Pattern P1 represents a program that accesses all its memory chunks with the same frequency. This is not realistic, but resembles the behavior of a memory stressing benchmark. Pattern P2 is more fitting to VNFs: some chunks are accessed frequently while others are only accessed seldomly. It is known that network traffic commonly has elephant flows and mice flows. If we imagine a router with a routing table in memory, the entries that are matched by elephant flows are accessed more often than mice flow packets. This could cause an access pattern like P2. We do not want to restrict ourselves to specific patterns, as there might also be patterns like P3 that do not have smooth transitions, but rather a step at some point.

In the figure we also sketched how this relates to the caches. As the caches mainly work in a least recently used manner (LRU), this pattern translates to hit rates of the caches. The hit rate of a cache measures what ratio of the accesses to the cache were served from the cache. Thus the hit rate of the LLC is proportional to the following formula:

$$LLC\ Hitrate \sim \frac{Area\ within\ Cache}{Area\ not\ within\ Cache}$$

In the figure we marked the area within cache as A1 and the area not within cache as A2 for pattern P1. The proposed scheduler influences how much LLC each VNF can use. In the depicted case the cache is more useful to the VNFs with P2 and P3 than the VNF with P1 as they have a higher access frequency in this region. The set of data that is used by the VNF is often referred as *working set*. The working set can be equal to the allocated memory, but it can also be smaller, e.g. if some data is only used for the initialization of the VNF. Further, the access pattern depends also on the network condition, e.g., an IDS serving highly diverse traffic has a different access pattern than an IDS that only filters a single connection [1].

The CPU utilization is the share of the CPU cycles where the CPU is active, i.e., not in a sleep state. Besides actual processing cycles the CPU is also active while waiting for data. Consequently, the CPU utilization increases for the same number of executed instructions if the $LLC\ Hitrate$ decreases.

In order to emulate different possible behaviors of VNFs, we propose to use simplified memory access patterns. From the definition of the access patterns it can be derived that the access frequency is monotonically decreasing with the sorted locations (horizontal axis). On the other hand due to the diversity of possible VNFs it is hard to find a more specific pattern. Therefor we propose a simple and generic model that is fully described with the allocated memory M, the maximum access frequency R and a distribution parameter α. The access pattern can be described with the function $r(m)$, which is the access rate at memory location m:

$$r(m) = R \cdot (\alpha + 1) \cdot (1 - m/M)^{\alpha} \ with\ m \in [0, M] \quad (1)$$

The parameter R models the packet rate, as the memory access rate of VNFs is proportional to the packet rate. M is the working set of the VNF. Finally parameter α describes a probability distribution that allows us to emulate a range of different access patterns. By setting $R = 1$ and $M = 1$ in $r(m)$, we get the underlying probability distribution, with the PDF $f(x)$ and the CDF $F(x)$:

$$f(x) = (\alpha + 1) \cdot (1 - x)^{\alpha} \quad (2)$$

$$F(x) = 1 - (1 - x)^{\alpha+1} \quad (3)$$

If we chose for example $\alpha = 0$, the distribution becomes uniform and we get an uniform access pattern with constant rate R for all memory positions $m \in [0, M]$. This kind of access pattern is also sketched in Figure 4 as P1 (solid line). For other values of the parameters R, M and α, we obtain different access patterns.

In this work, we only consider a quasi-static set of VNFs such that the access pattern $r(m)$ is not time dependent. This means that the VNFs and the traffic pattern of the VNFs does not change over time. This enables us to study the potential gains of LLC scheduling in deep.

V. Optimal Scheduler Design

In this section we first describe the developed algorithm which determines the optimal allocation for static access patterns.

```
output: $Allocation^*$
1  initialize: all cores share the LLC;
2  $W_c = 0\ \forall c$;
3  $Z = 1$;
4  $Allocation^* \leftarrow current\ allocation$;
5  while $\sum_c W < LLC_{tot} - 2$ do
6      $c^* = \arg\max_c(U_c)$;
7      if $W_{c^*}! = 0$ then
8          $W_{c^*} \leftarrow W_{c^*} + 1$;
9      end
10     else
11         $W_{c^*} \leftarrow \lfloor LLC_{c^*}/1.5MByte \rfloor$
12     end
13     if $\max(U_c) < Z$ then
14         $Z = \max(U_c)$;
15         $Allocation^* \leftarrow current\ allocation$;
16     end
17 end
```

Algorithm 1: Min-max scheduler determining the LLC allocation that minimizes the maximum CPU utilization

A. Optimization objective

There are different optimization objective possible. We imagine to use the scheduler in an NFV environment enabling scaling as described in Section III-A. In this environment the NFV orchestration would scale up, i.e., launch new instances, if some threshold is exceeded. As a result it makes sense to reduce the CPU utilization and thus making the scaling unnecessary, consequently saving resources. Hence, we choose to minimize the maximum CPU utilization of all cores. But the objective could also be, e.g., to minimize the sum of utilizations of all cores, or the reduction of memory access delays for selected VNFs. Furthermore, with a CPU utilization of 100%, packet loss is caused that should be avoided. Summarizing, a reduction of the maximum CPU utilization of all cores is desirable.

B. Algorithm

Algorithm 1 defines the algorithm that finds the LLC allocation which minimizes the maximum CPU utilization of all cores. The system is initialized with the default allocation, i.e. all cores compete for the LLC. W_c denotes the number of exclusively assigned cache ways of core c. No core has exclusive cache ways in the beginning (line 2). Before starting the algorithm loop, the upper bound for the minimal maximum CPU utilization of all cores Z is 1, as this is the maximum value of utilization. The algorithm also remembers the current allocation to cover cases where the initial allocation is already the optimal allocation (line 4). The following steps are repeated until all possible cache ways are distributed. First the core that has currently the highest CPU utilization is determined, this core is called c^*. If this core was already scheduled before, i.e., there are already cache ways exclusively assigned, the number of exclusive cache ways is increased by 1. Otherwise it is measured how much LLC the core currently

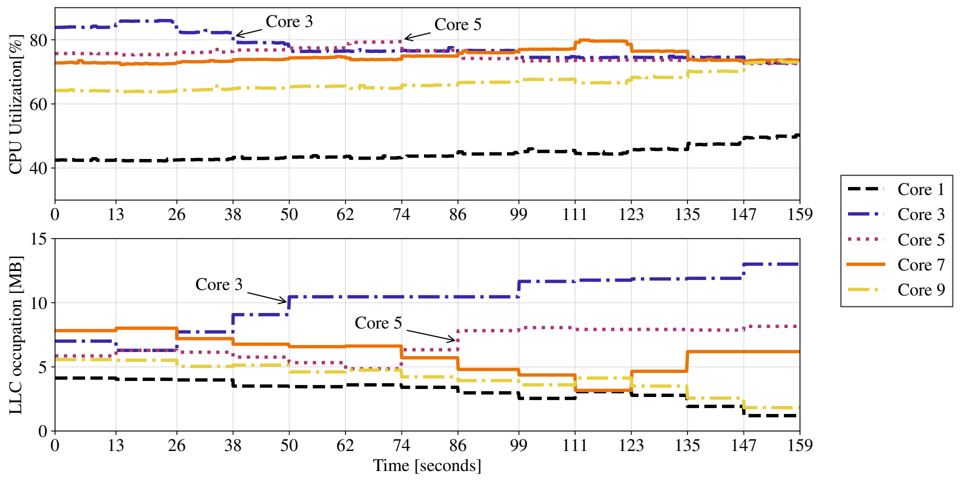

Figure 5. CPU utilization and LLC occupation of an exemplary scheduler run with 5 cores. The duration of each step is 12 seconds.

uses and this value is used to determine an initial number of ways. Each cache way corresponds to 1.5 MB of LLC, as the number of ways has to be integer the algorithm applies the floor operation. In most cases the LLC for the initialized core is increased in a subsequent step. The floor operation is a conservative choice in this case, as it guarantees that not too many cache ways are allocated.

After each schedule update, the algorithm checks if the step resulted in a new upper bound Z. In this case, the new bound and the new allocation are saved (lines 14 & 15).

The algorithm assigns at least one cache way in each step until all cache ways are assigned. In our set-up the LLC is 20-way associative, at minimum 2 ways must be left for cores without exclusive ways. This results in a maximum of 18 steps for the algorithm to find the optimal allocation. In general the algorithm is of linear complexity with the number of cache ways.

C. Example Run

Figure 5 shows one example run of the scheduler as presented in Algorithm 1. The upper graph shows the CPU utilization of the used cores, the lower one shows the LLC occupation of each core. The LLC Occupation metric measures how much LLC each core is currently using.

At time $t = 0\,s$ the system is initialized with the LLC shared by all cores.Before doing anything the scheduler determines the CPU utilization of all cores using PCM as described in section III-D.

In the first LLC allocation update at $t = 13s$, the highest core, in this case Core 3, is assigned exclusive ways. The number of ways for this first allocation is computed as in line 11 of Algorithm 1. As Core 3 can use less LLC than before, the CPU utilization of Core 3 is increased after this step. After this update, the scheduler waits until the CPU utilization has stabilized and measures again the CPU utilization of the Cores.

The next schedule update is done at $t = 26$ and increases the LLC of Core 3 by one way. After further updates at $t = 38$ and $t = 59$, the CPU utilization of Core 3 is below the utilization of Core 5, thus Core 5 is now scheduled in the updates at $t = 62$, $t = 74$ and $t = 86$.

This scheme continues until the optimum allocation is reached at $t = 147$ after 12 steps of the algorithm. It can be seen that we reached an absolute gain of 10.5% with respect to the maximum CPU utilization of all cores in this run.

D. Optimality Discussion

The scheduler minimizes the maximum CPU utilization of all cores for the static case. The CPU utilization of a core depends only on the allocated LLC if the memory access pattern is static. Thus the CPU utilization of core c is given with the function $U_c(LLC_c)$ where LLC_c denotes the share of the LLC usable by core c. The maximum CPU utilization of all cores is then:

$$f(\underline{LLC}) = \max_c(U_c(LLC_c)) \tag{4}$$

$\underline{LLC}$ is a vector with length C that denotes the current allocation of the LLC to the cores. Therefore, our optimization problem is:

$$\begin{aligned} \min \quad & f(\underline{LLC}) \\ \text{s.t.} \quad & \sum_c LLC_c \leq LLC_{tot}\ , \forall c \\ & LLC_c \geq 0\ , \forall c \end{aligned} \tag{5}$$

The constraints are due to the limitation of the total LLC of the chip LLC_{tot} and that the LLC allocated to one core must be non-negative.

The CPU utilization $U_c(LLC_c)$ is monotonically decreasing with LLC_c. More cache can only decrease the CPU utilization, as the CPU has to wait less for data. This means that the minimum f^* must be on the edge of the feasibility space, as we could otherwise increase LLC_c for any c and at least reach the same or a lower value of f.

The gradient of f, ∇f is a vector with $\nabla f_c = \frac{df}{dLLC_c}$. f in some point $\underline{LLC}^\circ$ is the maximum of the functions

Table I
NOTATION

Symbol	Description
$c \in \{0, ...C-1\}$	core number
C	number of cores of the CPU
LLC_c	LLC allocation to core c
$\underline{LLC} = \begin{bmatrix} LLC_0 \\ \vdots \\ LLC_{C-1} \end{bmatrix}$	LLC allocation of the CPU as a vector
$U_c(LLC_c) \in [0,1]$	CPU utilization of core c
$f(\underline{LLC}) = \max_c(U_c(LLC_c))$	Maximum CPU utilization of all cores
LLC_{tot}	Total Last Level Cache available

$U(LLC_c)$ in this point. It only depends on one dimension $c^\circ = \arg\max_c(U(LLC_c))$.

Consequently it holds:

$$\nabla f_c = \begin{cases} g, & \text{if } c = \arg\max_c(U_c(LLC_c)) \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

As all functions U_c are monotonically decreasing, we can state that $g \leq 0$. Note that we do not consider edge cases where the utilization of two cores is exactly equal, as they are obviously very rare in reality.

In each step the scheduler increases LLC_c for the core with the highest value and decreases it for the not scheduled cores. This means that the scheduler is moving along a line on the edge of the feasible space in every step. It always increases the LLC in dimension $c = \arg\max_c(U_c(LLC_c))$ and reduces it for the not scheduled dimensions $c^- \in NS \subset \{0, ...C-1\}$ and thus the scheduler is doing a gradient descent on the surface of the feasible space.

The step size is set to the size of one LLC way, which means we could overshoot in the case the step before was closer to the minimum. The scheduler takes this into account by saving the last valid Z (Z can never increase due to the conditions). Additionally, we argue that due to the characteristics of f and ∇f, in every point the final Z is close (within one step) to the global minimum f^*. We can not guarantee to reach f^* as the scheduler uses integer number of ways for scheduled cores.

VI. Evaluation

This section first discusses the experiment design used for evaluation. First, we present the results that indicate a transient phase of the LLC after an update. Secondly we show the scheduler gain depending on different parameters of the co-located VNFs.

A. Experiment Design

In order to evaluate the developed LLC scheduler, we evaluate 4000 scheduler runs. Depending on how many steps of the algorithm are needed each run has a duration of 150-200 s with one measurement point per second yielding an extensive data set.

Table II
EXPERIMENTAL PARAMETERS USED FOR EVALUATION.

Parameter	Range	Description
α	$\{0.3, 1.01, 2.5, 5\}$	Distribution parameter
M	$[1, 30]$ [MB]	Working set size
R	$[2000, 7000]$ $[s^{-1}]$	Access rate
C	5	Number of VNFs
U_c	$40\% < U_c < 90\%$	CPU utilization constraints

One example scheduler run is shown in Figure 5. In every run, 5 VNFs are active and each one running pinned to one core. As explained before we consider a static scenario, i.e. within one run, the CPU utilization is constant, if the LLC allocation is not changed. Obviously this assumption is not realistic in real deployments, as e.g. the packet rate changes continuously, but it enables us to analyze the scheduler gain and the inertial behavior of the CPU utilization after an LLC allocation update. We emulate VNFs using a C++ program running inside a VM virtualized with KVM. The program is allocating a table of size M and accessing the memory with rate R using the distribution given in Equation 3. The code for the emulation is published for reference [1].

In this work we want to research the achievable gain of LLC scheduling. Thus have chosen a wide range of parameters, shown in Table II in order to represent different kinds of VNFs and their interference. Even though only 5 distinct values of α where chosen, these values represent a wide range of access patterns.[2] E.g. $\alpha = 0.3$ is an access pattern where the complete working set is accessed almost uniformly, on the other hand with $\alpha = 5$ a large portion of the working set are accessed only seldom. Therefore larger or smaller values of $alpha$ do not change the overall behavior much and real VNFs fall somewhere in between.

Data that are accessed in VNFs are often state tables such as connection tables in a NAT or a firewall. One connection entry has to track the state of a connection and thus rather small. E.g. in the case of Linux' netfilter conntrack module [18] one entry has 376 Byte on our test server, thus the maximum of $M = 30$ MB corresponds to roughly 80000 concurrent connections. We do not expect a real VNF to have a significantly larger working set.

Further, the scenarios are generated such that they are in accordance with the NFV MANO architecture that is described in Section III-A. Hence, no VNF should be underutilized or overloaded. As a result, all VNFs in one scenario have a CPU utilization in the interval $[40\%, 90\%]$. The runs are generated as follows. The settings of each emulated VNF namely R and M are chosen randomly within the intervals shown in Table II, α is chosen randomly from the set shown. Afterwards the 5 chosen VNFs are executed on our measurement server and the CPU utilization is measured. If the CPU utilization of an VNF

[1]VNF emulation source code: https://github.com/tum-lkn/vnf-emu

[2]Note that $\alpha = 1.01$ is used, as otherwise with $\alpha = 1$ Equation 3 is much less complex to compute and thus the emulated VNF behaves different.

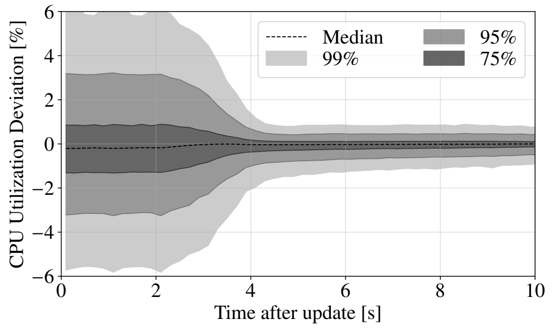

Figure 6. Transient phase after an LLC allocation update: Difference between the CPU utilization at a certain time after the update with respect to the median utilization of the CPU utilization in the interval [7,10].

is not within the defined interval R is increased or decreased. Then the CPU utilization is measured again. This pattern is repeated until the CPU utilization of all VNFs fall into the defined interval.

B. Transient phase of the LLC

As we are dealing with a real system, the CPU utilization is always not fully constant over time. Reasons for this can be, e.g., periodic tasks the operating system or the hypervisor is performing. More importantly, the CPU utilization shows a transient behavior after an update of the LLC allocation: Cache lines are only evicted if other data not cached yet, is accessed by the CPU. Furthermore, the CPU gain from LLC cache only shows if the cache line is accessed after that a second time, as only then the accessing delay is reduced.

Figure 6 visualizes this transient phase. Results were gathered from 500 scheduler runs with one measurement point per second, each scheduler run needs multiple steps and thus yielding multiple transient phases. We define the median of the CPU utilization in the interval [7,10] s after a reallocation of the LLC as baseline or true utilization after the update. Next we compute the difference of each measurement value with the baseline and show the distribution as a contour plot showing the percentiles of the outcomes. It can be seen that the CPU utilization can differ significantly from the baseline for the first four seconds, after this the CPU utilization clearly stabilizes.

In line 6 of Algorithm 1, the scheduler measures the current CPU utilization of all cores. As this measurement must not be influenced by the last iteration of the scheduler, the algorithm has to wait until the CPU utilization is in a steady state. As we are aiming for the optimal allocation, we used a conservative time of 12 seconds for each step. Though in real systems one might trade off the step time for a faster scheduler convergence.

C. Scheduler Gain

Next we evaluate how much gain can be expected from such an approach in a real system. In order to analyze this we evaluate 4000 different sets of 5 VNFs and determine

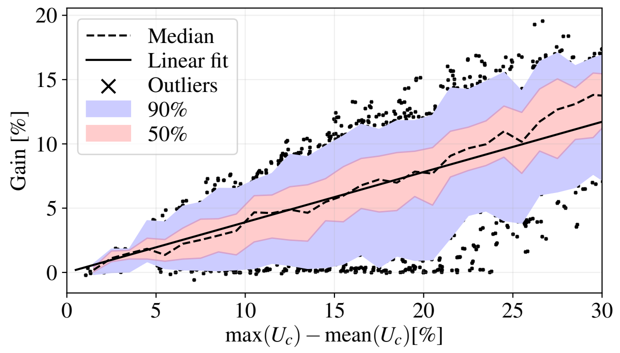

Figure 7. Relation between gain and difference between the CPU utilization of the highest core and the mean of the CPU utilization of all cores

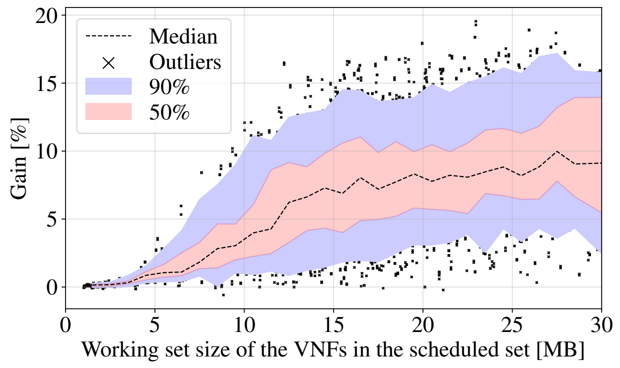

Figure 8. Relation between gain and the (mean) allocated memory of the scheduled VNF(s)

the optimal allocation with our scheduler. From this sets we compute the gain as:

$$Gain = \max(U_c)^{Shared} - \max(U_c)^{Scheduled}$$

where $\max(U_c)^{Shared}$ is the maximum CPU utilization with no LLC allocation (LLC is shared) and $\max(U_c)^{Scheduled}$ is the maximum CPU utilization in the optimized case.

Figure 7 shows the gain with respect to the difference between the maximum CPU utilization and the mean utilization for the shared allocation. The linear fit line shows that for every percent of difference between max and mean one can expect 0.4 percent more gain. In the extreme case, where the maximum is equal to the mean, the scheduler cannot achieve anything as a reduction of the utilization of one core increases the utilization of other cores. This means that a real system needs a certain degree of variability between the VNFs running on one server, otherwise no scheduling is possible. Consequently setups that deploy only equal VNFs on one server and additionally load-balance between the instances, such that the utilization is as well equal for all VNFs, cannot reduce CPU utilization with LLC scheduling.

On the other hand the outliers without gain show that this metric can not solely explain the achievable gain, but also depends on other metrics. One of these metrics is the allocated memory of the VNFs. The scheduler increases the LLC share for the core that has the maximum CPU utilization. If the complete working set already fits into the share of the LLC, a further increase of the allocated LLC does not yield any further gain. As a consequence it can be expected that LLC scheduling works worse for VNFs with a small working set. Thus we analyze the influence of the mean allocated memory of the scheduled set to the gain. We define the scheduled set as the VNF(s) that have exclusive ways in the final state of the scheduler. This means that each of the VNFs in the set is pinned to a CPU core which had the highest utilization in at least one scheduler interval.

Figure 8 shows how the allocated memory of the VNFs influence the gain. It can be seen that some minimum memory of around 5 MB is necessary to achieve gains. Between 5 and 12 MB the median gain quickly increases and flattens out for higher amounts of memory. Consequently VNFs that do not store much data, like for example a stateless firewall that only needs to access its ACL regularly cannot reduce their CPU utilization. On the other hand, due to the limited size of the LLC (the LLC is 30 MB in total), the achievable gain is also limited even for a large working set size. A VNF requiring 30 MB of data can never cache everything in the LLC as this would leave the other VNFs without cache, which is technically impossible due to restrictions in the CPU chip architecture.

VII. Conclusion

In this work, we show an optimal LLC scheduler for NFV environments. We show that LLC scheduling can reduce the maximum CPU utilization on one NFV server significantly. How much gain can be achieved depends on the employed VNFs and their traffic. As the CPU utilization of one VNF is decreased by increasing the allocated LLC of this VNF, the other VNFs that are co-located on the same server can use less LLC. Consequently this increases the CPU utilization of the co-located VNFs. Thus in order to enable the scheduler to reduce the overall maximum CPU utilization the difference between mean and maximum CPU utilization of all cores must not be too small.

Furthermore we show that the working set size of the scheduled VNFs must be large enough. Otherwise LLC scheduling cannot bring any gains.

The developed scheduler algorithm does only consider static cases, i.e. traffic and VNFs must not change. This assumption does not old in many real applications, as network traffic is known to be dynamic over time. Thus, in the future we want to extend the algorithm to also be able to work in more dynamic environments that also consider changes in the network traffic. Finally in this work, we consider only one NFV server, while NFV was introduced to work with a large pool of servers. Thus this work could also be extended to consider larger deployments and an interplay with a VNF placement strategy.

Acknowledgments

This work has been partially funded by the German Research Foundation (DFG) under the grant numbers KE1863/6-1 and KE 1863/8-1.

References

[1] C. Zeng, F. Liu, S. Chen, W. Jiang, and M. Li, "Demystifying the Performance Interference of Co-Located Virtual Network Functions," in *IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[2] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[3] "Qualcomm Centriq™ 2400 Processor." [Online]. Available: https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf

[4] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling slos in network function virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[5] Y. Hu, M. Song, and T. Li, "Towards "Full Containerization" in Containerized Network Function Virtualization," *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 467–481, 2017.

[6] P. Veitch, E. Curley, and T. Kantecki, "Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation," in *IEEE Conference on Network Softwarization (NetSoft)*, 2017.

[7] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, ser. NSDI'12, 2012.

[8] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gomez, "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology," in *IEEE/ACM Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[9] S. G. Kim, H. Eom, and H. Y. Yeom, "Virtual machine scheduling for multicores considering effects of shared on-chip last level cache interference," in *IEEE International Green Computing Conference, IGCC 2012*, 2012.

[10] R. Nathuji and A. Kansal, "Q-Clouds : Managing Performance Interference Effects for QoS-Aware Clouds," in *5th European conference on Computer systems (EuroSys)*, 2010.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[12] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015.

[13] A. R. Nath, A. Kansal, S. Govindan, J. Liu, and Suman, "PACMan: Performance Aware Virtual Machine Consolidation," in *10th IEEE International Conference on Autonomic Computing (ICAC'13)*, 2013.

[14] G. ETSI, "Network functions virtualisation (nfv): Architectural framework," *ETsI Gs NFV*, vol. 2, no. 2, p. V1, 2013.

[15] M. Beierl, "NFV-KVM-Tuning," 2016. [Online]. Available: https://wiki.opnfv.org/display/kvm/Nfv-kvm-tuning

[16] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar, "Elastic network functions: opportunities and challenges," *IEEE Network*, vol. 29, no. 3, 2015.

[17] P. F. Thomas Willhalm, Roman Dementiev, "Intel Performance Counter Monitor - A better way to measure CPU utilization," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[18] P. Ayuso, "Netfilter's connection tracking system," *LOGIN: The USENIX magazine*, vol. 31, no. 3, 2006.