ResQ: Enabling SLOs in Network Function Virtualization

Amin Tootoonchian, *Intel Labs;* Aurojit Panda, *NYU, ICSI;* Chang Lan, *UC Berkeley;* Melvin Walls, *Nefeli;* Katerina Argyraki, *EPFL;* Sylvia Ratnasamy, *UC Berkeley;* Scott Shenker, *UC Berkeley, ICSI*

https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

ResQ: Enabling SLOs in Network Function Virtualization

Amin Tootoonchian[⋆] Aurojit Panda[¶‡] Chang Lan[†] Melvin Walls[§]
Katerina Argyraki[•] Sylvia Ratnasamy[†] Scott Shenker[†‡]

[⋆]*Intel Labs* [†]*UC Berkeley* [‡]*ICSI* [¶]*NYU* [§]*Nefeli* [•]*EPFL*

Abstract

Network Function Virtualization is allowing carriers to replace dedicated middleboxes with Network Functions (NFs) consolidated on shared servers, but the question of how (and even whether) one can achieve performance SLOs with software packet processing remains open. A key challenge is the high variability and unpredictability in throughput and latency introduced when NFs are consolidated. We show that, using processor cache isolation and with careful sizing of I/O buffers, we can directly enforce a high degree of performance isolation among consolidated NFs – for a wide range of NFs, our technique caps the maximum throughput degradation to 2.9% (compared to 44.3%), and the 95th percentile latency degradation to 2.5% (compared to 24.5%). Building on this, we present ResQ, a resource manager for NFV that enforces performance SLOs for multi-tenant NFV clusters in a resource efficient manner. ResQ achieves 60%-236% better resource efficiency for enforcing SLOs that contain contention-sensitive NFs compared to previous work.

1 Introduction

Modern networks are replete with dedicated "middlebox" appliances that perform a wide variety of functions. In recent years, operators have responded to the growing cost of procuring and managing these appliances by adopting Network Function Virtualization (NFV). In NFV, middlebox functionality is implemented using software Network Functions (henceforth NFs), which are deployed on racks of commodity servers [18, 36, 38]. This approach offers several advantages including lower costs, easier deployment, and the ability to share infrastructure (*e.g.,* servers) between NFs.

However, there is one oft-overlooked disadvantage to the move to software. Because physical instantiations of these functions relied on dedicated hardware; they had well-understood performance properties which allowed operators to offer performance SLOs [3, 7, 44]. Providing performance guarantees is harder with software, particularly when multiple NFs are *consolidated* on the same server.

While current NFV solutions [28, 41, 47, 49] typically place NFs on dedicated cores, this is insufficient to ensure *performance* isolation. Even when run on separate cores, NFs share other processor resources such as the last-level cache (LLC), memory, and I/O controllers (Figure 1),

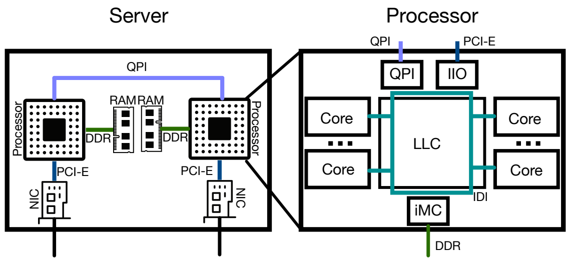


Figure 1: *High-level view of shared resources inside a server and CPU.* A typical NFV deployment consists of racks of servers interconnected with a commodity fabric. Each server consists of a set of resources (CPU, RAM, NIC) interconnected with standard interfaces (QPI, DDR, PCIe). A modern general-purpose Intel CPU consists of a number of processor cores all sharing the *uncore* that includes I/O controller (IIO), integrated memory controller (iMC), last-level cache (LLC), and in-die interconnect (IDI).

collectively referred to as *uncore* resources [29]. NFs contend for these uncore resources and, as we show, such contention can degrade an NF's throughput by as much as 40% compared to its performance when run in isolation (§2).

Providing performance guarantees in NFV essentially boils down to solving the *noisy neighbor* problem, common in multi-tenant environments [62]. Traditionally, this problem has been addressed through resource partitioning. However, in the NFV context, performance variability primarily stems from contention for the LLC [10] and, until recently, no mechanism existed to partition the LLC.[1] This changed with the introduction of processor features – *e.g.,* Intel® Cache Allocation Technology (CAT) [27] – that provides hardware mechanisms for partitioning the LLC across cores.

CAT is a mechanism that opens the door to a new approach for performance isolation in NFV. However, this mechanism has neither been widely tested in nor applied to the NFV context. Hence, in this paper, we study whether and how CAT can be applied to support performance SLOs for NFV workloads. More specifically, we explore the following two questions.

First, we evaluate whether CAT is *sufficient* to ensure performance isolation across NFs? We show that CAT "out of the box" does *not* provide predictable performance: instead, some NFs' performance continues to vary (by as much as 14.7%) depending on their neighboring NFs. This contradicts

[1] Instead, prior work on providing SLOs aimed to *predict* the impact of contention on performance [10]. However, such prediction is difficult and, as we show in §6.2, is no longer accurate with newer hardware and software.

Application	Description	Mpps	Instructions/Cycle	L3 refs/Packet	L3 hit rate	Kilocycles/Packet
Efficuts [61]	Efficuts classifier (32k rules)	1.224	0.63	10.23	99.92	1.72
EndRE [1]	Click-based WAN optimizer	3.770	1.95	1.54	99.95	0.56
Firewall	Click-based classifier (250 rules, sequential search)	0.366	0.59	1.59	99.44	5.74
IPsec	Click-based IPsec tunnel using IPsec elements	0.442	3.31	5.13	99.83	4.75
LPM	Click-based IP router pipeline with RadixIPLookup	5.475	1.92	3.87	99.80	0.38
MazuNAT	Click-based NAT pipeline by Mazu Networks	2.698	1.53	12.14	99.92	0.78
Snort [53]	Inline IDS (20k rules [16, 57]) with netmap for I/O	0.683	1.94	25.61	97.03	3.06
Stats	Click-based flow stat collection with AggregateIPFlows	3.685	1.28	10.22	99.92	0.57
Suricata [46]	Inline IDS (20k rules [16, 57]) with netmap for I/O	0.205	1.61	26.89	98.36	10.
vEPC	Standalone software implementation of LTE core network	-	-	-	-	-

Table 1: *Characteristics of NFs used in this work.* The performance is measured when the NF is run alone with exclusive access to 45 MB LLC with a test traffic of min-sized packets sampled from a pool of 100k flows uniformly at random. Snort and Suricata use netmap while the rest use DPDK for I/O. We do not report these statistics for vEPC due to constraints discussed in §4.1.

prior work [10, 63] which identified cache contention as the main source of performance variability for NFs. Careful investigation reveals the cause of this problem: poor buffer management with *Intel Data Direct I/O* [31], a processor feature that enables direct NIC-to-LLC transfers (*i.e.,* bypassing memory), can lead to a *leaky DMA* problem in which packets are unnecessarily evicted from LLC leading to variable performance. We describe a simple buffer sizing policy that avoids the leaky-DMA problem and show that, with this policy, CAT *is* sufficient to ensure performance isolation (with performance variability under 3% across all scenarios).

Next, having ensured the robustness of CAT as a performance isolation knob given proper buffer sizing, we turn to the question of how to *apply* CAT in a practical system. The challenge here lies in designing a scheduler that assigns resources to NFs in a manner that is accurate (no SLO violations), efficient (minimizing resource use), and scalable (so that decisions can be easily adapted to changing workloads and infrastructure).

We develop ResQ, a cluster resource scheduler that provides performance guarantees for NFV workloads. ResQ computes the number of NF instances required to satisfy SLO terms, and allocates LLC and cores to them. ResQ balances accuracy and efficiency by first *profiling* NFs to understand how their performance varies as a result of LLC allocation. For scalability, ResQ uses a fastpath-slowpath approach. We formulate the scheduling problem as a mixed-integer linear program (MILP) that minimizes the number of machines to guarantee SLOs. Solving this MILP optimally is NP-hard and hence ResQ uses a greedy approximation to schedule NFs upon admission. In the background, it periodically computes a near-optimal solution, and only moves from the greedy to this solution when doing so would lead to a sufficiently large improvement.

We show that ResQ is accurate (with zero SLO violations in our test scenarios), efficient (achieving between 60–236% better resource efficiency compared to prior work based on prediction [10]) and scalable (can profile and admit new SLOs in under a minute).

To our knowledge, our work is the first to analyze the efficacy of using CAT to solve the noisy neighbor problem for a wide range of NFs and traffic types, and ResQ is the first NFV scheduler to support performance SLOs, showing that the benefits of NFV need not come with the loss of what has traditionally been a vital part of carriers' service offerings. ResQ is open source and the code can be found at `https://github.com/netsys/resq`.

The remainder of this paper is organized as follows: we start by quantifying the impact of contention on NFV workloads (§2) and then provide relevant background information and elaborate on the problem we address (§3). We study whether CAT is sufficient for performance isolation in §4, then present the design and evaluation of ResQ in §5 and §6 respectively. We discuss related work in §7, and finally conclude.

2 Motivation

A reasonable first question to ask is whether the current NFV approach of running multiple NFs on shared hardware results in performance variability, *i.e.,* does the *noisy neighbor* problem matter in practice for NFV workloads. We address this question by evaluating the effects of sharing resources for a range of NFs (listed in Table 1), and by comparing their throughput and latency when they are run in *isolation* – *i.e.,* on a dedicated server with no other NFs – to their performance in a *shared* environment comprising of a mix of 11 other NFs (see §4.1). In both cases, we run the NF being evaluated on its own core and allocate the same set of resource to it, thus avoiding any contention due to core sharing. We repeat our measurements using both small (64 B) packets and large (1518 B) packets, and send sufficient traffic to saturate NF cores. We delay a more in depth discussion of our experimental setup to §3.

We show the results of our comparison in Figure 2, which shows the percent degradation in throughput and 95th percentile latency. Each bar shows the maximum performance loss for an NF running on shared infrastructure when compared to the isolated run. We observe that 7 of the NFs we test demonstrate a performance degradation of more than 10%, while another 5 show a degradation of more than 20%. Some suffer significant throughput (up to 44.3%) and latency degradation (up to 24.5%) and we find that this holds for both small and large-packet workloads. We also tested the effect of contention on a virtual Evolved Packet Core (vEPC) system using a domain specific packet generator, and observed an 80% degradation in throughput. The vEPC packet generator does not measure latency, and as a result we do not include these results in Figure 2. Finally, we expect that NF degradation will worsen as we increase the number of NFs that share a server.

In conclusion, we find that most NFs suffer significant degradation due to resource contention – this holds for both small and large packet traffic.

3 Background and Problem Definition

Next we present some background for our work, focusing in particular on describing NFV workloads, identifying sources of contention that affect network functions, evaluating prior work in this area, and introducing the processor cache isolation mechanism used by ResQ. Finally, we define the NFV SLO enforcement problem that we address in the rest of this paper.

3.1 NFV Workloads

NFV workloads consist of packet-processing applications, canonically referred to as Network Functions (NFs); they range from relatively simple with lightweight processing (*e.g.,* NAT, firewall) to more heavyweight ones (*e.g.,* vEPC [17]). NFs may be *chained* together such that packets output from one NF is steered to another. For example packets might first be processed by a firewall and then a NAT.

NF performance can vary – even in the absence of the noisy neighbor problems, and an individual NF running in isolation will often display variance in performance across runs. Work over the last decade has led to practices that have been shown to improve performance stability for software packet processors and are now widely understood and adopted [11, 13, 42]. The most significant ones include running NFs on dedicated and isolated cores that use local memory and NICs (NUMA affinity), maintaining interrupt-core affinity, disabling power saving features (*i.e.,* idle states, core and uncore frequency scaling), and disabling transparent huge pages. We adopt the same and, from here on, all our discussion of performance predictability assumes that the above techniques are already in use. As we shall show, these are necessary but not sufficient – we still need to address contention for shared resources, which is our focus in this paper.

3.2 Sources of Contention

Naïvely, one might believe that placing NFs on independent cores ensures that they do not share resources.[2] However, in modern processors, cores share several resources. Resources shared across cores include: PCI-e lanes and CPU's integrated I/O controller, and memory channels and CPU's integrated memory controller, and last-level cache (LLC) as shown in Figure 1. Currently, most servers do not oversubscribe PCIe lanes, and NICs do not contend for these resources. While independent NFs might share PCIe lanes when sharing NICs using SR-IOV [33] or through a software switch [25], one can control contention for these resources by rate limiting ingress traffic received by an interface. As a result the main resource that NFs in shared infrastructure can compete on are memory and LLC, and we study the effect of both in this paper.

[2]There may also be contention within the fabric connecting different servers; that is outside the scope of this paper, but we envisage that standard fabric QoS and provisioning mechanisms [6, 54] can be applied there.

3.3 Prior Work (or Lack Thereof) in NFV

To our knowledge, the only work that analyzes the impact of resource contention on NFV workloads is a work by Dobrescu *et al.* [10]. That work proposes using a simple model for predicting performance degradation due to contention for the last level cache. However, as shown in §6.2, the model is inaccurate when tested under newer hardware and different workloads – *e.g.,* we find that their model overestimates the impact of contention by as much as 13% (a relative error of 75%) for newer hardware and workloads. In addition, that work focuses on predicting degradation rather than meeting performance guarantees; consequently, it does not discuss how one can *enforce* a desired limit on the level of contention. In contrast, our work focuses on enforcing SLOs using hardware mechanisms such as CAT. As we show in §4, ResQ provides robust performance guarantees for a variety of workloads.

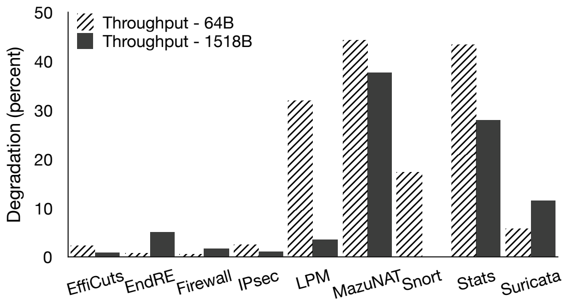
Other work has looked at managing NFV jobs. This includes works such as E2 [47], Stratos [20], OPNFV [36]. While these systems perform some basic allocation of resources to NFs, none consider contention nor do they aim to provide performance guarantees. ResQ can be incorporated into these systems allowing them to provide performance guarantees; ResQ is currently under evaluation for adoption in one commercial orchestrator.

3.4 Hardware Cache Isolation

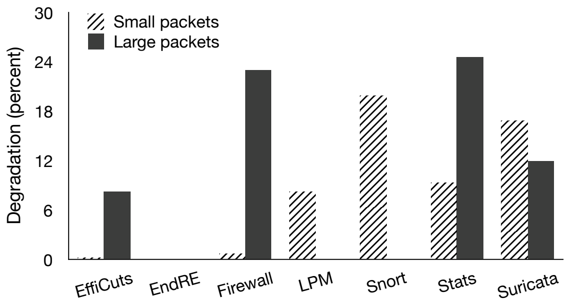
ResQ's enforcement relies on recent processor QoS features implemented in processors that enable monitoring and control of shared processor resources. For Intel processors, these features are collectively known as the Intel Resource Director Technology (RDT) which include Intel Cache Allocation Technology (CAT) [30][3] and Cache Monitoring Technology (CMT). They allow users to allocate or monitor the amount of cache accessible to or used by threads, cores, or processes.

To monitor a set of processes or cores using CMT, the kernel allocates a resource monitoring ID (RMID) which the processor uses to collect usage statistics for them. The kernel

[3]Cache partitioning is also available in other server processors, for example Qualcomm's Amberwing processor [43] which is based on ARM64.



(a) Maximum throughput drop



(b) Maximum 95th percentile latency increase

Figure 2: *Maximum performance degradation for minimum and MTU-sized packets without isolation.* Due to interference, throughput and latency degrade up to 44.3% and 24.5% respectively. Small and large packet trends are similar for all NFs. We do not measure latency for NFs that mangle packets in a way incompatible with our traffic generator's timestamp embedding (MazuNAT, IPSec), and vEPC whose domain-specific traffic generator does not report latency. LPM and Snort do not exhibit sensitivity with large packets due to a testbed limitation: their cores were not saturated at line rate.

updates a core register with this RMID upon context switch to bind the monitored entities to the RMID. Similarly, when limiting the cache available to processes or cores, the kernel first allocates a class of service identifier (CLOS). It then updates a register to specify the amount of cache accessible to a CLOS. Finally, the kernel can associate a CLOS with a process by updating the appropriate register when switching to the process. Linux allows users to specify the set of processes to be monitored using a newly introduced `RDT` interface. For the evaluation reported in this paper, we used these features as implemented on the Xeon® processor E5 v4 family which allows users to specify up to 16 cache classes. The processor we use for our evaluation allows us to enable access to between 5%–100% of the cache, in 5% increments, for each CLOS. To our knowledge ResQ is the first research work that uses CAT to provide performance isolation in NFV.

3.5 Problem Definition

Our goal is to support performance SLOs for NFV workloads. The conjecture driving this paper is that CAT gives us a powerful and practical knob to achieve this. To validate this conjecture we must answer the following questions:

1. The crux of providing SLOs is knowing how to isolate different NFs from a performance standpoint. Is CAT sufficient to ensure performance isolation between NFs or do we also have to consider contention for other resources? We study this question in §4.
2. CAT is ultimately just a configuration knob and using it in a practical system raises a number of questions: what is a good scheduling algorithm that balances scalability (scheduling decisions per second), accuracy (minimizing SLO violations), and efficiency (minimizing use of server resources)? What is the API for SLOs or *contract* between NFs and the NFV scheduler? What information do we need from NFs to make good scheduling decisions? We address these through the design, implementation, and evaluation of ResQ in §5 and §6.

4 Enforcing Performance Isolation

Dobrescu *et al.* [10] argued that the level of LLC contention entirely determines NF performance degradation. This observation would lead one to believe that merely enabling CAT – which controls the level of cache contention – is sufficient to ensure performance isolation, *i.e.,* ensure that one NF's performance is unchanged due to the actions of any other colocated NF. In this section we evaluate this hypothesis, and find that it does not hold; we then explain why this is the case and present our strategy for mitigating this issue.

4.1 Experimental Setup

NF workloads. We ran our evaluation on a range of NFs (see Table 1) including: NFs from the research community (*e.g.,* Efficuts [61], EndRE [1]) and industry (*e.g.,* Snort [53], Suricata [46], vEPC [17]); NFs with simple (*e.g.,* Firewall, LPM) and complex (*e.g.,* Snort, Suricata) packet processing; NFs with small (*e.g.,* IPSec) and large (*e.g.,* Snort, Stats) working set sizes; NFs using netmap [52] (*e.g.,* Snort and Suricata) and DPDK [12] (*e.g.,* Efficuts and Click) for I/O; NFs that are standalone (*e.g.,* Snort) and those that are built on frameworks like Click (*e.g.,* MazuNAT). We also evaluated the impact of contention on an industrial virtual Evolved Packet Core [17] system[4] that implements LTE core network functionality in software. Due to licensing issues these tests were run on a different testbed, and made use of a domain-specific commercial traffic generator.

Test setup and CAT configuration. We ran all our evaluation on a server with an Intel Xeon E5-2695 v4 processor and dedicated 10 Gb/s and 40 Gb/s network ports.

We repeat the same experiments as in §2 after enabling CAT. We evaluate two scenarios for each NF:

- *Solo* run, where we run the NF under test on a single core and CAT is configured to allocate 5% of LLC to the NF (the smallest allocation with CAT). We run *no* other NFs run on the machine. This provides us with a baseline for

[4]Vendor name anonymized due to licensing requirements.

how the NF behaves with a specific LLC allocation but no contention on the other resources.

- *Shared* runs, where the NF under test is run on a single core and we use CAT to allocate 5% of LLC to the NF. We run 11 instances of a different competing NF on the remaining cores, these instances share the remaining 95% of LLC. We repeat this experiment to analyze the performance impact of each type of competing NF, *i.e.,* in each iteration we pick a different NF from Table 1 to use as the competing NF.

Observe that in both cases, the NF under test is allocated the same number of cores and the same amount of LLC. In our experiments we measure the target NF's performance in terms of throughput and 95th percentile latency, and compute performance degradation for *shared runs* compared to a *solo run*.

NF	Size (bytes)	Degradation (%)	LLC Miss Rate (%) NF	TX	RX	Mem BW Util. (%)
MazuNAT	64	2.4	65.5	0.02	8.49	31
	1518	12.2	72.9	55.3	88.6	94
Stats	64	0.3	64.5	0.07	8.84	31
	1518	14.7	73.1	63.5	89.1	99

Table 2: *CAT does not sufficiently isolate NFs in shared runs with large packets.* The culprit is the high memory bandwidth utilization with large packets which in turn is because the "leaky DMA" issue renders DDIO ineffective. All numbers are the worst-case numbers for shared runs. TX/RX LLC miss rate and memory bandwidth utilization are processor-scoped whereas NF degradation and its LLC miss-rate are NF-scoped.

4.2 Is CAT Sufficient?

Surprisingly, our results were mixed and showed that CAT is *not always* sufficient for providing performance isolation: while CAT was successful at isolating NFs when processing small (64 B) packets, where throughput and latency degradation remained below 3%, it could not isolate all NFs when large (1518 B) packets were used: we observed degradation of up to 14.7% for some NFs (*e.g.,* MazuNAT and Stats). This is particularly surprising in light of the fact that NFs process packets at a higher rate when small packets are used, as a result, memory accesses should be more frequent for smaller packets compared to larger packets, and one would expect greater degradation for smaller packets.

We began our investigation into this anomalous result by checking whether there was a difference in cache miss rates between shared and solo runs. Unsurprisingly, we found no noticeable difference and concluded that CAT was functioning as expected. Next, we analyzed measurements from other hardware counters in the platform and found that memory bandwidth utilization increased substantially in going from small to large packets (Table 2).

Can memory contention affect NF performance? To answer this question, we first used the Intel Memory Latency Checker (MLC) [32] to measure memory access latency as a function of increasing memory bandwidth utilization. We plot the memory access speed (*i.e.,* inverse of the latency)

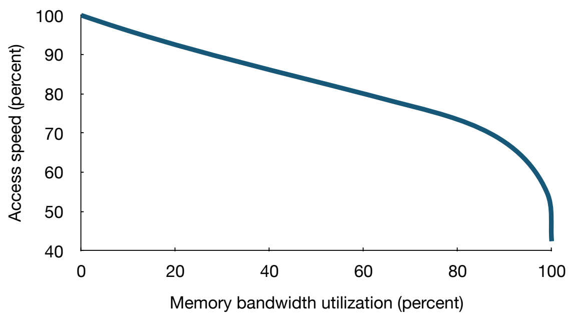


Figure 3: *Memory access speed as a function of load.* The memory access latency increases linearly with load on memory controller up to around 90% utilization.

in Figure 3 and find that with up to approximately 90% load on memory channels, the memory access speed degrades linearly with increase in load, and subsequently experiences super-linear degradation dropping to 40% of the baseline value.

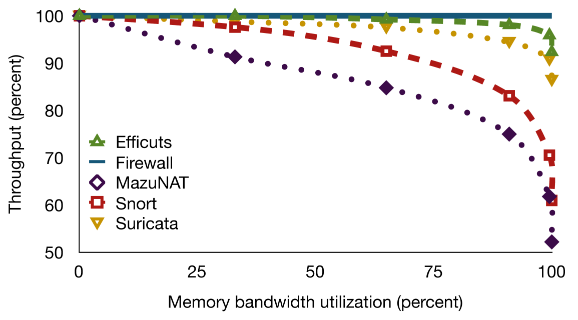


Figure 4: *Normalized NF throughput for a selection of NFs as a function of memory load.* The curves track the memory access speed curve (Figure 3) very closely.

Next, we checked whether this observation meant that NF performance would also degrade with increased memory contention. We analyzed this by running NFs under the same environment as was used for the solo runs and running MLC on the other cores of the same server to generate memory bandwidth load. We show the results for this experiment in Figure 4 and find that the added memory contention does lead to performance degradation for NFs; NFs like MazuNAT are up to 50% slower with aggressive memory contention. Note that this is a near worst-case degradation in response to memory contention – MLC exhibits a more aggressive memory access pattern when compared to network functions (and most other applications).

What causes memory contention? We certainly did not expect to see much memory traffic in our shared workload. While a single core is capable of inducing around 12 GB/s traffic on the memory controller, we expect cores running NFs to generate a fraction of this load. That is because, cycles during which the NF may access state are spaced out by cycles spent on compute intensive portions of packet processing including I/O, framework processing, and stateless portions of the NF processing. To empirically validate this hypothesis, we wrote a synthetic NF that accesses DRAM 1000 times per packet and observe that it can only generate 2.5 GB/s of memory traffic – we expect a realistic NF to generate far less traffic.

Furthermore, our processor is equipped with Intel Data Direct I/O (DDIO) technology [31] which lets DMAs for packet I/O interact with the last level cache rather than going to DRAM. As a result, we did not expect packet I/O to contribute to memory contention. However, given our expectation that NF state accesses should be more frequent with small packets (due to higher packet rates) we suspected that some interaction with DDIO might be the root cause of the substantial increase in DRAM traffic.

4.3 The Leaky DMA Problem

By default, DDIO is limited to using 10% of the LLC. When a buffer that needs to be DMAed is not present in LLC, it is first brought into the LLC resulting in memory traffic. We hypothesized that this might be the cause of memory contention in our system. Furthermore, DMA transfers are mediated by the processor DMA engine (as opposed to a core), therefore, cache misses during DMA are not included in the core-based LLC counters we used when evaluating the efficacy of CAT above. To test our hypothesis, we looked at CPU performance counters that measure PCIe-sourced LLC references and misses, and found that the I/O-related LLC miss-rates increased from nearly 0% to around 60% on the TX path and 90% on the RX path when going from small packets to large packets in the shared runs. This showed that DDIO is ineffective at preventing memory contention in our system, but why?

The LLC space used by DDIO cannot be partitioned using CAT, and is shared across NFs. As a result, if the aggregate number of packet buffers exceed DDIO's LLC space then packet I/O can contend for cache space and evict buffers holding packets being processed. The maximum number of in-flight packets is bounded by the number of descriptors available to NIC queues. For the experiments above, NFs had their own queue each with 2048 descriptors – that is a total of 24576 buffers for 12 queues. In the shared runs, this translates to requiring 3 MB of cache space for small packets (each spanning 2 cache lines), but a whopping 37.5 MB for large packets (each spanning 25 cache lines).

As noted earlier, we had not observed significant changes in NF cores' cache miss rates when comparing solo and shared runs. This suggested that LLC contention due to DDIO does not affect parts of the packet that are processed by the NF. We thus found that DDIO frequently evicts cache lines belonging to packets that are being processed, and these are needed soon after eviction for packet TX. Similarly the RX path frequently needs to fetch buffers that were previously evicted due to DDIO space contention. Together, they result in much of the network traffic and stale buffers to bounce back and forth between LLC and DRAM multiple times. We refer to this problem as the *leaky DMA* problem and identify it as the root cause of performance variability for NFs when CAT is used.

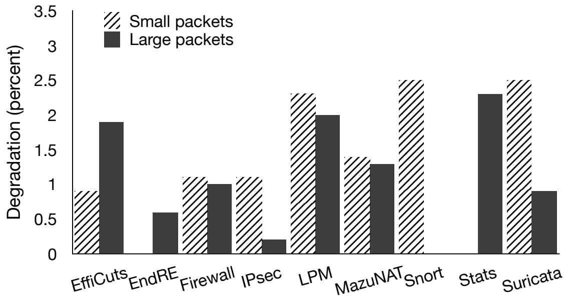
4.4 Solution: CAT + Buffer Sizing

Fortunately, both DPDK and netmap provide mechanisms to control the number of DMA buffers used by the system. In case all DMA buffers are in use, no packets are received from the NIC. DMA buffers become available once the packet data contained within them is freed, at which point new data can be received. Therefore controlling the number of in-flight buffers allows us to control the efficacy of DDIO. In ResQ, we restrict the size of the pool from which packet buffers are allocated based on the aggregate number of MTU sized packets that can fit in the LLC space reserved for DDIO, thus avoiding the leaky DMA problem. Note that NICs also contain a sizeable buffer (4096 packets in Intel NICs) and as a result this restriction does not result in packet loss unless the incoming link is congested.

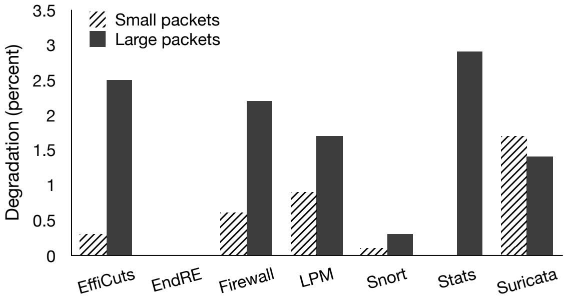
We evaluated the efficacy of using buffer sizing to solve the *leaky DMA* problem by rerunning both solo and shared runs after fixing the number of allocated packet buffers to the number calculated above. As shown in Figure 5, this resulted in a situation where for both packet sizes throughput and latency degradation were less that 3% (including the vEPC which is not reported in the Figure 5), thus confirming our fix.

Other Resources. Given our experience with memory contention, one might be concerned about contention on other resources which we briefly discuss here. We do not observe notable *IOMMU* contention [50] since we use statically-mapped DMA buffers backed by huge pages. The maximum degradation we observe in a microbenchmark that maximizes core to *IDI* traffic is below 4% – in practice, the IDI utilization is much lower and the degradation is negligible. *IIO* throughput in the Haswell/Broadwell processors is around 160 Gbit/s which may introduce a bottleneck if all PCIe lanes (40) are more than half utilized. However, the aggregate traffic per CPU remains below 150 Gbit/s in our experiments; classical QoS mechanisms would sufficiently address the fair-sharing of this resource. Consequently, we conclude that contention for these other resources is not a concern given the current architecture.

Recap. To summarize we found that while memory contention can be a source of performance variability, this is *not* a result of NF behavior, but rather because of poor DMA buffer sizing which can result in the *leaky DMA* problem. The leaky DMA problem causes DMA buffers to be repeatedly evicted from cache which in turn results in high memory bandwidth utilization. We address the leaky DMA problem by appropriately controlling the aggregate number of active DMA buffers, and find that this, in conjunction with CAT, is sufficient to ensure performance isolation for NFV workloads. Fi-



(a) Maximum throughput drop



(b) Maximum 95th percentile latency increase

Figure 5: *Maximum degradation in throughput and 95th percentile latency for minimum and MTU-sized packets when target NF is isolated per §4.4.* They are both consistently below 3% across all the experiments. When comparing, note the difference in the y-axis range with Figure 2 and that the results do not include latency measurements for IPSec and MazuNAT due to traffic generator's constraints.

nally, our results here show that as opposed to what was found by Dobrescu *et al.* [10], issues beyond LLC sizing can affect NF performance. We believe this is because of changes to software (*e.g.,* use of DPDK), and hardware architecture (*e.g.,* DDIO). Additionally that work considered only small packet sizes and did not analyze accuracy with MTU-sized packets.

5 ResQ

In this section we present the design and implementation of ResQ, a cluster resource manager that is designed to efficiently schedule NFs while guaranteeing that SLOs hold. We begin by describing ResQ's design and we focus our discussion on three aspects:

Service interfaces: Traditionally, network operators have relied on resource overprovisioning to meet performance objectives with hardware network appliances. NFV can allow us to guarantee SLOs while more efficiently utilizing resources. However, achieving this greater efficiency requires that tenants provide ResQ with workload and other information in addition to the NF. We describe ResQ's inputs and the types of guarantees it can enforce in §5.1.

NF profiling: Given an input NF, ResQ needs to determine its resource requirements. These depend on the NF configuration, input traffic, and platform and thus varies across tenants and operators. In §5.2, we describe ResQ's efficient and automatic profiler that measures how NF performance (both throughput and latency) varies as a function of LLC allocation. The ResQ profiler minimizes the number of executions required to collect this information, and can thus rapidly profile a large set of NFs. The profiler's output is a key input to the ResQ scheduler.

NF scheduling: Finally, we present our scheduler in §5.3. ResQ implements a two-level scheduler that takes as input NFs, SLO specifications and requirements, and profiling results and determines *(a)* the number of NF instances to start, *(b)* the server(s) on which these instances must be placed; and *(c)* the amount of the LLC to assign to each instance.

5.1 ResQ SLOs

How do we improve efficiency of resource utilization while continuing to meet performance objectives? Our insight is that how an NF performs – given a fixed set of resources – depends on two factors. First, NF configuration such as rule set of a firewall or an IDS – the size and complexity of this configuration directly affects performance [4, 15]). Second, traffic profile which captures characteristics such as distributions for flow arrival, flow sizes, packet sizes, and packet interarrivals. NF data sheets often highlight that performance depends not just on the input traffic rate but also on factors such as the number of new sessions per second and traffic mix [48]. ResQ improves scheduling efficiency by accounting for these factors when allocating resources.

Tenants can specify two types of performance SLOs: *reserved* and *on-demand*, which we explain next.

Reserved SLOs specify the NF or chain, its expected configuration and traffic profile, and its performance target (*i.e.,* expected latency and throughput). Given this information, ResQ profiles (see §5.2) the NF to determine its performance as a function of resource use, and uses this information to allocate resources. ResQ does not distinguish between NFs or chains of NFs and profiles a chain similarly to a single NF. Since we assume that the traffic profile, configuration, and maximum input rate (specified as part of the performance target) do not vary, implementing the computed allocation is sufficient to satisfy the SLO term. Run-time deviations from the specified traffic profile or NF configuration may only violate the corresponding SLO term – it does not affect other SLOs because ResQ provides sufficient isolation among SLOs. Tenants are required to submit a new admission request to ResQ in the event any of these parameters change; in response, ResQ may either reallocate resources or deny admission if objectives cannot be met.

ResQ ensures stable resource usage for NFs making use of reserved SLOs. This simplifies resource provisioning for the network operator without significantly affecting efficiency for NFs with stable configuration and input

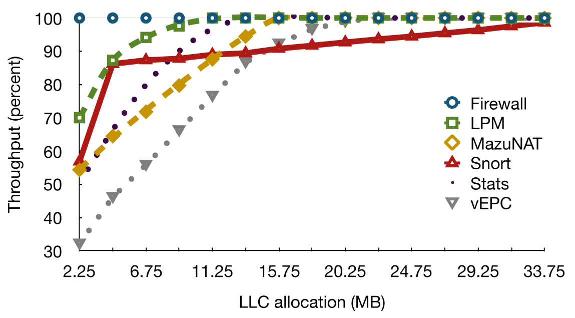


Figure 6: *Normalized throughput as a function of LLC allocation for a selection of NFs.*

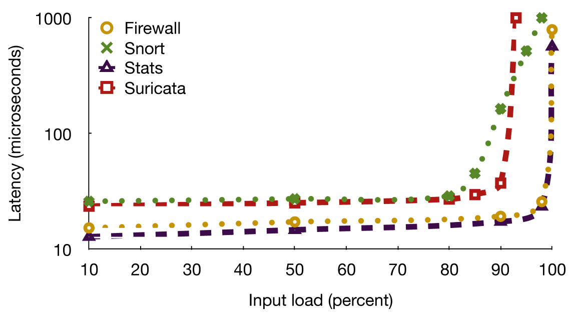


Figure 7: *Latency as a function of normalized input traffic load for a selection of NFs.*

traffic. We envision that, similar to cloud providers, network operator will encourage the use of reserved SLOs by providing volume discounts to tenants. Reserved SLOs are, however, inefficient for NFs with highly variable workloads – *e.g.,* NFs with high traffic variance – since these must be overprovisioned to meet worse-case traffic demands. Such NFs are better suited to use on-demand SLOs.

On-demand SLOs specify the NF and target latency. ResQ continuously monitors NF latencies and resource utilization, and dynamically adjusts resource allocations to meet the target latency independent of the input traffic or configuration. If the target latency could not be met under a best-case allocation, ResQ raises an error. Furthermore, ResQ relies on traffic policing to appropriately reduce input load if it is unable to meet the total traffic demand – *e.g.,* due to lack of resources or reaching a user-specified cap on resource usage. We provide further details about in §5.3.

5.2 ResQ Profiler

ResQ relies on performance profiles to determine resource allocation for reserved SLOs. A ResQ profile consists of throughput-LLC allocation (*e.g.,* Figure 6) and latency-load (*e.g.,* Figure 7) curves. To construct these curves, the profiler runs a set of experiments and collects measurements. The time taken to run one experiment varies depending on the traffic pattern – it takes around 5 seconds with our sample traffic profiles. Building a general NF profile that is valid across all configurations and traffic patterns would likely require exploring a potentially unbounded space and is infeasible. Profiles generated by ResQ are, therefore, specific to not just the NF, but also the configuration and traffic pattern specified by a reserved SLO. Since our profiles are quite specific, we might require a large number of profiles for an NFV cluster; consequently, we must ensure that profile generation is *fast*. Furthermore, errors in an NF profile affect ResQ's accuracy and efficiency, and therefore we need to ensure that generated profiles are *accurate*. We rely on interpolation, with dynamically varying interpolation intervals, to quickly produce accurate profiles as described below.

The throughput-LLC allocation curve for an NF can be generated by running it alone on a profiling server and measuring its throughput as the profiler varies the amount of allocated LLC using CAT §3.4. To generate the latency-throughput curve (*e.g.,* Figure 6), the profiler launches the NF with a given LLC allocation and measures the 95th percentile latency as a function of input traffic rate. This measurement is repeated for different LLC allocations to produce a latency-throughput curve. In Figure 7, we show an example of such curves for a fixed LLC allocation (we chose to allow NFs to access all of the LLC in this case).

Since the profiler is in the critical path of the admission control process, naïvely running all the required experiments (400 datapoints for around 20 utilization levels and 20 LLC allocations) delays the process significantly (*e.g.,* 34 minutes with 5 second runs). To alleviate this bottleneck in the admission control process, we observe that these curves could be accurately constructed with far fewer datapoints.

We observed that, across a wide range of NFs, the latency-throughput curves vary only slightly for different LLC allocations. As a result, we can safely approximate this curve by measuring an NF's worst-case latency, which corresponds to the LLC allocation that maximizes NF throughput. Furthermore, we observed that both sets of curves are monotonically increasing, and that in all cases the throughput-LLC allocation is concave, while the latency-throughput curve is convex. This allows us to approximate the curve by measuring throughput and latency at a few points, and using linear interpolation to compute values for intermediate points. We implement our interpolation as follows: the profiler begins by measuring the minimum, maximum and midpoint of each curve. It then computes the linear interpolation error by comparing the interpolated value with the measured mid-point. If the interpolation error is above 1%, the profiler recursively splits both intervals and repeats the same procedure. The profiler stops collecting additional measurements once the interpolation error falls below 1%. In our experience, each profile required between 8–12 measurements and could be constructed in under a minute.

5.3 ResQ Scheduler

The ResQ scheduler is comprised of two parts:

- A *centralized scheduler* is responsible for admission control, placement for all SLOs, resource allocation for reserved SLOs, and setting aside resources on individual servers for on-demand SLOs.

- A *server agent* that runs on each server and is responsible for configuring the server, monitoring resource utilization, and detecting SLO violations. The server agent implements a *local scheduler* that is responsible for allocating resources to NFs with on-demand SLOs that are placed on the server by the centralized scheduler.

In clusters running ResQ, tenants submit SLO requests to the centralized scheduler which performs admission control. For on-demand SLOs, the scheduler checks if the cluster has sufficient resources available to launch one instance of the NF (the supplied NF description includes information about minimum resources required by an instance), and rejects the SLO should sufficient resources not be available. For reserved SLOs, the scheduler consults the NF profile (see §5.2) to determine whether the SLO is feasible; if so, the scheduler uses a greedy algorithm (see §5.3.1) to compute NF placement and resource requirements for meeting the performance objectives. Admission is denied if the greedy algorithm cannot find a fit, otherwise it notifies the appropriate server agents to launch NF instances and allocate the requested resources to them. The scheduler also programs the datacenter fabric so as to steer traffic to these NF instances – similarly to existing NFV schedulers [20, 21, 47, 51], we assume that the fabric will split traffic across these instances.

While the greedy allocation computed by the central scheduler is sufficient for meeting the performance objectives, it might not be optimal in terms of resource use. Therefore, in the background, ResQ also periodically solves a mixed-integer linear program (MILP) to find a (near-)optimal schedule. If the gap in resource usage between this and the greedy schedules exceeds a configurable threshold, ResQ migrates running NFs[5] to implement the optimal schedule. Migrating to the optimal schedule frees up more resources that can be used to accommodate other SLOs.

On-demand SLOs are scheduled locally by server agents. Upon submission of an on-demand SLO, the centralized scheduler finds a server that has sufficient resource to run one instance of the NF and assigns the on-demand SLO to that server. The server agent uses max-min fair allocation to partition the on-demand LLC space among such NFs. If the server agent is unable to meet the NFs latency targets, it notifies the central scheduler which in turn adds NF instances to the cluster.

Next we provide more details about the algorithm used for scheduling both types of NFs.

5.3.1 Reserved SLOs

Computing the optimal schedule for reserved SLOs is an NP-hard problem. Hence, we develop an online greedy algorithm for fast admission. After the profile is generated, ResQ attempts to greedily bin-pack the NF instances using a first-fit heuristic, which works as follows.

1. It divides the target throughput by the expected throughput of a single instance to estimate the number of NF instances required to meet the objective. The expected throughput of one instance is what a single instance can sustain when allocated a fair share of LLC (*i.e.,* the available LLC divided by the number of cores) such that its latency does not exceed the target latency.
2. It calculates the minimal LLC allocation for each instance by iteratively adding a unit of LLC allocation to each instance in a round-robin fashion until the aggregate throughput is above the target.
3. It places instances on servers using the first-fit decreasing heuristic, *i.e.,* places the largest instance first. If this algorithm succeeds, ResQ launches the instances each with the computed schedule.

The greedily generated schedule may be suboptimal because *(a)* it is online and incremental (does not move running instances), and *(b)* uses a heuristic to determine how many NF instances to run. To improve the placement efficiency, in the background, ResQ computes an optimal schedule. We formulate the placement problem as a mixed-integer linear program whose objective is to minimize the number of servers used (see Appendix A). We use a MILP solver [24] to compute the (near-)optimal schedule. In our experiments, the greedily and incrementally computed schedule's resource use is within 20% of the optimal one (see §6).

The solver typically finds near-optimal solution(s) for inputs which require a cluster size of around 40 servers in seconds to minutes. To scale to larger-sized clusters, we partition the SLOs into sets and pass each to a different solver. The computed schedules are instantiated on different slices of the cluster. This allows us to trade off computation time for schedule optimality.

The computed MILP-based schedule might be different than the running schedule that was greedily updated during admission. To converge to the new schedule various NFs must be migrated; this problem has been studied in the literature in the form of migration of stateful middleboxes or scaling out NFs [21, 51, 55]. This is likely an expensive and disruptive process, therefore we migrate only when the optimality gap is large enough.

Alternatively, a *migration avoidance* [47] strategy could be deployed to avoid the disruption or complexity of state migration. This involves booting up new instances but leaving old instances (that were to be terminated) running – the old instances will continue serving their traffic but no new traffic is directed to them. When their traffic eventually dies down they will be terminated. This strategy is only effective when sufficient spare capacity is available to bring up new instances without terminating the old ones.

[5]We rely on standard VM migration techniques.

5.3.2 On-Demand SLOs

Resource allocation for on-demand SLOs is jointly performed by the local resource scheduler and the centralized scheduler. The central scheduler is responsible for leasing dedicated cores and LLC space to local schedulers for scheduling on-demand SLOs, and for assigning new NF instances to servers with spare resources. Leases are dynamically adjusted when reserved SLOs arrive or leave the system – *i.e.,* beyond configured resources reserved for on-demand SLOs, the central scheduler may make spare resources available to local schedulers.

When computing LLC allocations for on-demand SLOs, NFs are placed in a shared LLC space or dedicated partitions based on whether or not their latency objectives can be met with sharing. Sharing LLC space (when possible) helps minimize the overhead of LLC partitioning since CAT allocates LLC space in fixed and relatively large increments. The NFs that require isolation are put in separate classes. If, despite isolation, the local scheduler fails to meet an NF's latency objective, it notifies the central scheduler which in turn adds more instances or resources for the failed SLO if possible.

The above-mentioned LLC allocation is computed as follows. First, all on-demand NFs are placed in a cache class that includes all the on-demand LLC space leased to the local scheduler. The server agent waits for a period of time for NFs to serve traffic before monitoring SLO violations and LLC occupancies (using Intel CMT – see §3.4) – occupancy measurements are used to estimate NFs' LLC demand. If one or more SLO violations are observed, the local scheduler continues with a max-min fair allocation of the LLC space. SLO-compliant NFs and SLO-violating NFs with low cache miss rates that use less than their fair share of LLC are put in a shared cache class with an allocation closest to the sum of their LLC occupancy. SLO-violating NFs with high cache miss rates that use less than their fair share of LLC are put in an isolated cache class with an allocation close to their fair share. The rest remain in the shared cache class whose size is reduced to the remaining on-demand LLC space. This procedure is repeated for the shared cache class to completion.

The server agent is responsible for monitoring on-demand NF instances for SLO violations. Such violations may occur when traffic pattern or NF configuration changes. Upon detection of a violation or change in the leased LLC space size, the local scheduler repeats the LLC allocation procedure to find whether it could meet the new demand with local resources and if not would notify the central scheduler of the new failed SLO.

6 Evaluation

In this section, we address the following questions:

Accuracy: To what extent do contention-agnostic schedulers violate SLOs? We compare against a simple bin-packing strategy adopted by current contention-agnostic schedulers [20, 45, 47]. We compare accuracy both without CAT and when we use CAT to evenly partition LLC across NFs.

Efficiency: We compare the efficiency of ResQ's online (greedy) and offline (mixed integer program based) schedulers against a prediction-based online scheduler [10] and an E2-like scheduler [47] which dynamically scales the number of NF instances in response to input traffic load.

To answer these questions, we generated three sets of reservation-based SLOs each with around 200 terms: one involving only the cache-sensitive NFs; one involving only the cache-insensitive NFs; and one involving a mixture of all the NFs. We set the target throughput and latency for each SLO to 90% and 100% of what a single instance of the corresponding NF can sustain when run in isolation without LLC contention. To avoid interfering with DDIO's reserved LLC space (10%), ResQ uses only 90% of the available LLC. To enable comparison with the fair allocation scheme, we use 9 cores per server so that each core can be allocated an equal cache partition (10%).

6.1 Accuracy

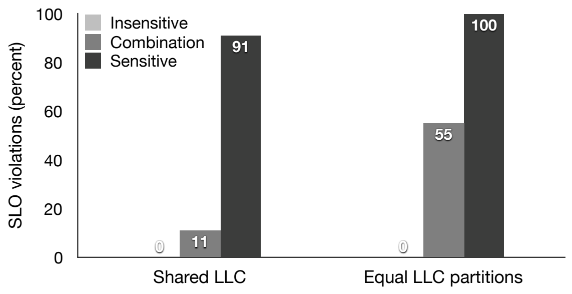


Figure 8: *SLO violations with* $>5\%$ *error for contention-agnostic methods.* Contention-agnostic placement results in throughput and latency SLO violations. As expected, violations increase with the sensitivity of NFs. Naïvely partitioning LLC has an adverse effect.

If SLO violations were rare, it would be appealing to opt for a simpler contention-agnostic scheduler. To assess this choice, we evaluate the ability of current contention-agnostic schedulers to meet SLOs. To do so, we first run each NF on a dedicated server without restricting cache access to determine its throughput and latency. We then use this information to pack NF instances on the first available server. We show the results in Figure 8. Unsurprisingly, no SLO violation are observed for the cache-insensitive workload. However, SLO violations are common for combination (11%) and cache sensitive workloads (91%) workloads.

Next, to check whether a naïve cache isolation strategy is sufficient to reduce violations, we reran the same workload after using CAT to partition the LLC evenly between all NFs on a server. The number of violations worsened in this case: 55% of SLOs are violated in the combination workload, while all SLOs are violated for the cache-sensitive case. In the combination case, this difference is due to the unavailability of the underutilized dedicated LLC space of the cache-insensitive NFs to the cache-sensitive ones.

Note that schedules computed by ResQ have no violations in all cases.

6.2 Efficiency

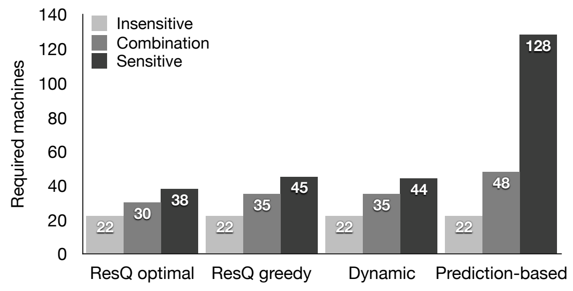


Figure 9: *Resource efficiency of different schemes and SLO mixes.* Not surprisingly, all methods are similar when NFs are cache-insensitive. The ResQ's greedy admission is within 19% of the optimal solution. The prediction-based scheme uses significantly more servers because it overestimates degradation.

ResQ builds on availability of a hardware mechanism (cache isolation) to provide predictable performance regardless of contention. Two alternative strategies for getting predictable performance involve: *(a)* online scheduling where one measures NF performance and dynamically allocates NF instances in response to SLO violations, and *(b)* using a performance predictor (*e.g.,* Dobrescu *et al.*'s predictor [10]) to predict throughput degradation due to resource sharing and using its result for scheduling. We analyze ResQ's efficiency in contrast to these options next.

ResQ's efficiency. For reserved SLOs, ResQ implements both an offline MILP-based scheduler that computes near optimal schedules and an online greedy scheduler. In Figure 9, we first evaluate the accuracy gap between these options. The optimal scheduler performs up to 19% better than the greedy scheduler. However, as previously noted in §5.3.1, the optimal scheduler may take much longer than the greedy scheduler, and ResQ can opportunistically move to using the optimal schedule if warranted.

Comparison with elastic scaling. Systems like E2 [47] continuously monitor NFs and dynamically add new instances if demand could not be met. A major drawback of this approach is that it cannot be used to enforce any *latency SLOs*. Despite being dynamic, this approach is not significantly more efficient than ResQ as seen in Figure 9. The dynamic approach uses the same number of instances for both cache-insensitive and combined traffic. It does provide a small savings of 1 machine (*i.e.,* a 2.2% improvement) for cache-sensitive workloads. However, this saving comes at the cost of no SLO isolation (variations in an NF's behavior may affect all its neighboring SLOs) and no latency guarantees.

Comparison with the predication-based approach. Finally, one could use a performance predictor to predict degradation due to resource sharing; this produces a safe schedule assuming the predictor never underestimates degradation. Dobrescu *et al.* [10] previously proposed such a predictor. Their predictor works as follows. Each NF is profiled using a series of synthetic benchmarks with tunable pressure on the LLC. The result is a curve which one can use to determine throughput as a function of competing LLC references. The competing LLC references are approximated by counting the LLC references of NFs' solo runs. This method was reported as being very accurate in 2012.

To study its robustness against significant hardware and software changes, we reran the experiments on our testbed using similar NFs and setup (6 competing NFs and 19.5 MB of LLC). Figure 10 shows the average prediction error in percentage points. Each bar shows the difference between predicted and observed performance drop suffered by a target NF when sharing a processor with 5 identical competing application instances (9 different sets of NFs for each NF) similarly to their choice of competitors. We find that this predictor is conservative and consistently overestimates degradation by a large margin. Consequently, it can be used to enforce (throughput) SLOs albeit not efficiently.

We use this predictor to build an online first-fit bin-packing scheduler. The scheduler packs an instance on the first server whose existing SLOs do not get affected by the new instance; it proceeds to pack a second instance if the predicted throughput is below the target throughput. We ran all the computed placements and recorded the *real* throughput and latency to assess SLO compliance. All schedules remain SLO compliant regardless of cache sensitivity except the prediction-based scheduler that violates 0.5% latency SLOs in the combination case. This is not surprising because this method does not predict latency degradation.

In Figure 9, we compare the efficiency of ResQ with the prediction-based method – by efficiency, we mean the number of CPUs (equivalent to servers for single-CPU servers) each scheduler needs to satisfy its SLOs. With

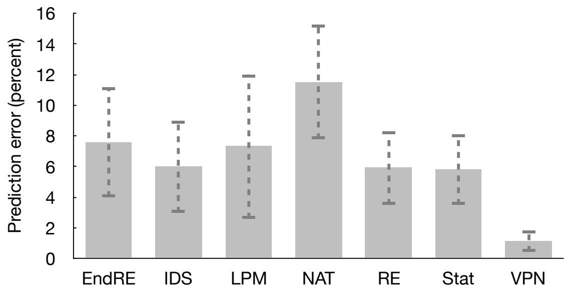


Figure 10: *Error of the throughput degradation prediction method [10].* We observe that errors are significantly higher using the current generation of hardware than what was previously observed. To follow the original setup, we use the following chains: EndRE is LPM → Stats → EndRE, VPN is LPM → Stats → IPsec, IDS is Snort, RE [58] is LPM → Stats → RE, STAT is LPM → Stats, NAT is MazuNAT, and IP is LPM.

cache-insensitive NFs, all schedulers need the minimum number of CPUs because consolidation does not affect the performance. As expected, the efficiency gap widens as the cache sensitivity of the mix of SLO NFs increases. The gap between the prediction-based scheduler and ResQ's greedy scheduler increases from 37.1% to 184.4%.

The are two reasons for such a sharp increase in resource usage for the prediction-based scheduler: *(a)* overly conservative performance estimate results in more false positives (mispredicting violations), and *(b)* lack of a mechanism to predict how much traffic an NF can handle without SLO violations. The gap in the sensitive case is due to the latter reason: individual servers have spare capacity but the scheduler cannot use any because an NF serving maximum traffic will violate the existing NFs' SLOs, but what if it only serves 20% of its capacity? These issues aside, scheduling is much simpler in ResQ because isolation is enforced by hardware regardless of contention.

Based on this result, we conclude that ResQ's simple first-fit bin-packing heuristic using CAT (online admission) is effective in maintaining a resource efficient and SLO-compliant schedule, while there is opportunity to further optimize this schedule by periodically running a slow offline scheduler.

7 Related Work

Performance modeling. Prior work [5, 9, 23, 40, 60] has investigated modeling and predicting the effect of resource contention in the context of HPC and datacenter applications. These models are often useful in contexts where the relative performance of two settings needs to be compared, *e.g.,* when scheduling or placing jobs. However, they are not accurate enough for our purposes. Dobrescu *et al.* [10] have proposed using cache references as a predictor of throughput with contending processes. While this was highly accurate given the hardware and software stacks available at the time, we find that it consistently underestimate throughput (§6.2) in today's systems. We showed that a scheduler using this predictor may consume up to 3× more resources compared to ResQ (§6.2). Moreover, this work on prediction models still leaves open the question of *enforcement* wherein an NF that deviates from its predicted behavior (whether due to malicious behavior, configuration changes, or varying traffic) can impact the performance of its neighboring NFs.

Performance isolation. Packet processing and NFV platforms [28, 41, 47, 49] do not isolate NFs from contending on uncore resources. Such systems can be extended to use CAT to provide performance isolation. Our contribution lies in showing how cache isolation can be used to both provide performance isolation and guarantee SLOs. Other systems that provide end-to-end performance guarantees for multi-tenant networks [2, 39, 56] treat CPUs as independent resource units and do not account for interference across cores. DRFQ [22] models a packet-processing platform as a pipeline of resources where each packet is sequentially processed by each resource. DRFQ's primary goal is to provide per-flow fairness while we focus on SLO guarantees. Ginseng [19] presents an auction-based LLC allocation mechanism, but does not offer SLOs. Heracles [37] uses CAT and other mechanisms to co-locate batch and latency-sensitive jobs while maintaining millisecond time-scale latency SLOs; we target more aggressive latency SLOs (high throughput, microsecond scale).

Mechanisms. The mechanisms and use cases of cache partitioning have been studied in the past [8, 34, 35]. A rich body of literature looks at software-only methods for cache isolation [14, 26, 59, 64]. Their performance implications have not been studied in the NFV context but they may be used as an alternative to CAT when hardware support is not available or more granular allocations are desired. A recent work [63] has also briefly looked at the benefit of using CAT to alleviate a specific instance of the noisy neighbor problem. It focuses on a single workload and demonstrates that, in one specific case, CAT notably improves performance in presence of a noisy neighbor problem. By contrast our work is general (covering a wide range of NFs and workloads), identifies cases where CAT alone does not sufficiently isolate NFs, and develops a contention-aware scheduler that uses our isolation mechanism to provide SLO guarantees for NFs.

8 Concluding Remarks

Despite no algorithmic innovation, ResQ's simple greedy scheduler achieves a significantly higher resource efficiency than prior prediction-based methods and its efficiency is on-par with elastic schedulers that do not guarantee SLOs. Moreover, despite its hardness, ResQ's MILP formulation yields (near-)optimal schedules in a matter of seconds to minutes. These advances were all made possible because we identified a technique – building on hardware cache isolation and proper buffer management – that ensures strong performance isolation regardless of noisy neighbors. ResQ is open source and available at `https://github.com/netsys/resq`.

Acknowledgement

We would like to thank Andrew Herdrich, Edwin Verplanke, Priya Autee, Christian Maciocco, Charlie Tai, Rich Uhlig, Michael Alan Chang, Yashar Ganjali, David Lie, Hans-Arno Jacobsen, our shepherd Tim Wood, and the NSDI reviewers for their comments and suggestions. This work was funded in part by NSF-1553747, NSF-1704941, and Intel corporation.

References

[1] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-system Redundancy Elimination Service for Enterprises. In *NSDI*, 2010.

[2] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[3] H. Basilier, M. Darula, and J. Wilke. Virtualizing Network Services- The Telecom Cloud. Ericsson Review, 2014. URL: `http://tinyurl.com/j5adfts`.

[4] Y. Beyene, M. Faloutsos, and H. V. Madhyastha. SyFi: A Systematic Approach for Estimating Stateful Firewall Performance. In *PAM*, 2012.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[6] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633.

[7] Broadband Forum. TR-178: Multi-service Broadband Network Architecture and Nodal Requirements, 2014. URL: `http://tinyurl.com/z7vkk6h`.

[8] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *ISCA*, 2013.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, 2013.

[10] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-processing Platforms. In *NSDI*, 2012.

[11] T. L. K. Documentation. Reducing OS Jitter Due to Per-CPU kthreads. URL: `http://tinyurl.com/mpnf4m3`.

[12] Data Plane Development Kit (DPDK), 2015. URL: `http://dpdk.org/`.

[13] DPDK Performance Tuning Guide, 2016. URL: `http://tinyurl.com/jkngtok`.

[14] 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 *ASPLOS*, 2010.

[15] S. Ehlert, G. Zhang, and T. Magedanz. Increasing SIP firewall performance by ruleset size limitation. In *PIMRC*, 2008.

[16] Emerging Threats. Emerging Threats Open Rulesets, 2016. URL: `http://tinyurl.com/nppr7ut`.

[17] The Evolved Packet Core. URL: `http://tinyurl.com/hvkukyw`.

[18] ETSI. Network Functions Virtualisation. URL: `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[19] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-Driven LLC Allocation. In *USENIX ATC*, 2016.

[20] A. Gember-Jacobson, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR*, abs/1305.0209, 2013.

[21] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[22] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource Fair Queueing for Packet Processing. In *SIGCOMM*, 2012.

[23] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *SOCC*, 2011.

[24] Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual, 2015. URL: `http://www.gurobi.com`.

[25] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[26] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS*, 2009.

[27] 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 *HPCA*, 2016.

[28] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI*, 2014.

[29] Intel® Xeon® Processor E5 and E7 v4 Families Uncore Performance Monitoring, 2016. URL: `http://tinyurl.com/zpsj63k`.

[30] Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, 2016. URL: `http://tinyurl.com/hasjlm2`.

[31] Intel® Data Direct I/O (DDIO), 2014. URL: `http://tinyurl.com/jlkzvll`.

[32] Intel® Memory Latency Checker, 2015. URL: `http://tinyurl.com/kgroxnw`.

[33] I. L. A. Division. PCI-SIG SR-IOV Primer, 2011. URL: `http://tinyurl.com/kt7bwqb`.

[34] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS*, 2004.

[35] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS*, 2007.

[36] Linux Foundation. OPNFV, 2016. URL: `https://www.opnfv.org/`.

[37] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[38] D. Lopez. OpenMANO: The Dataplane Ready Open Source NFV MANO Stack. In *IETF Meeting Proceedings, Dallas, Texas, USA*, 2015.

[39] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *NSDI*, 2015.

[40] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO*, 2011.

[41] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[42] Performance Tuning for Mellanox Adapters. URL: `http://tinyurl.com/y8slm66k`.

[43] T. P. Morgan. ARM Servers: Qualcomm is Now a Contender. `https://www.nextplatform.com/2017/08/23/arm-servers-qualcomm-now-contender/`, 2017.

[44] Nokia. Solutions: Residential Services Delivery, 2016. URL: `http://tinyurl.com/h3cwqsy`.

[45] T. L. Foundation. ONAP: Open Network Automation Platform. `https://www.onap.org/` retrieved 09/21/2017.

[46] Open Information Security Foundation. Suricata: Open Source IDS/IPS/NSM engine, 2015. URL: `http://suricata-ids.org/`.

[47] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[48] P. A. Networks. PA-3000 Series Datasheet. `https://www.paloaltonetworks.com/products/secure-the-network/next-generation-firewall/pa-3000-series` retrieved 09/21/2017.

[49] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[50] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir. Utilizing the IOMMU Scalably. In *USENIX ATC*, 2015.

[51] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*, 2013.

[52] L. Rizzo. Revisiting Network I/O APIs: The Netmap Framework. *ACM Queue*, 10(1), 2012.

[53] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[54] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[55] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *SIGCOMM*, 2015.

[56] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[57] Sourcefire's Vulnerability Research Team. VRT Rule Set, 2015. URL: `https://www.snort.org/talos`.

[58] N. T. Spring and D. Wetherall. A Protocol-independent Technique for Eliminating Redundant Network Traffic. In *SIGCOMM*, 2000.

[59] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *ASPLOS*, 2009.

[60] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *CGO*, 2012.

[61] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*, 2010.

[62] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *CCS*, 2012.

[63] P. Veitch, E. Curley, and T. Kantecki. Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation. *NetSoft*, 2017.

[64] X. Zhang, S. Dwarkadas, and K. Shen. Towards Practical Page Coloring-based Multicore Cache Management. In *EuroSys*, 2009.

A MILP Formulation

When a new reserved SLO is submitted, ResQ profiles the given NF (or chain) and, if admissible, greedily schedules one or more instances of it. Periodically, ResQ looks for a more optimal schedule to switch to if this results in significant resource savings. We formulate this optimal scheduling as a mixed-integer linear program.

Symbol	Type	Description
θ_i	Constant	Target throughput of SLO i
π_{il}	Constant	Pivot point of piece l of SLO i
α_{il}	Constant	Slope of piece l of SLO i
β_{il}	Constant	Y-intercept of piece l of SLO i
τ	Constant	Number of cores per machine
φ	Constant	LLC size per machine
U_i	Constant	Maximum input load of SLO i
I_{ijk}	Binary Var	Instance j of SLO i is assigned to machine k
N_k	Binary Var	Machine k is active
C_{ij}	Integer Var	LLC allocated to instance j of SLO i
λ_{ijl}	Binary Var	Piece l of instance j of SLO i is used

Table 3: *List of symbols used in MILP.* We use indices i, j, k, l for SLO terms, instances, machines, and profiles' linear fit pieces respectively. The number of variables and constants depend on the size of the cluster, number of SLO terms, maximum number of instances per SLO term, and number of pieces of individual profiles.

The objective of the MILP in Listing 1 is to minimize the number of machines used to satisfy all the SLO terms. As input, it expects system configuration and profiles, and produces a schedule as output. For each SLO term, this schedule provides the number of instances to start, where each instance should be placed, and the amount of LLC allocated to each instance. We encode the SLO profiles in the form of piecewise linear approximations of their throughput-LLC curves.

$$\min \sum_k N_k$$

$$\text{s.t.} \quad \sum_k I_{ijk} \le 1 \qquad \forall i,j \quad (1)$$

$$\sum_{i,j} C_{ij} . I_{ijk} \le \varphi \qquad \forall k \quad (2)$$

$$N_k \le \sum_{i,j} I_{ijk} \le N_k . \tau \qquad \forall k \quad (3)$$

$$\theta_i \le \sum_j [\mathrm{U}_i . \sum_l \lambda_{ijl} . [\alpha_{il} . C_{ij} + \beta_{il}]] \qquad \forall i \quad (4)$$

$$\sum_l \lambda_{ijl} . \pi_{il} \le C_{ij} \le \sum_l \lambda_{ij(l+1)} . \pi_{i(l+1)} \qquad \forall i,j \quad (5)$$

$$\sum_l \lambda_{ijl} = \sum_k I_{ijk} \qquad \forall i,j \quad (6)$$

Listing 1: *Mixed-integer linear program that minimizes the number of machines used to meet reserved SLOs in ResQ.* A brief description of the symbols appear in Table 3.

We use a set of variables to capture the scheduling results and constants to encode the system configuration and profiles:

θ_i specifies the target throughput for SLO term i.

π_{il} specifies the pivot point for piece l of the throughput-LLC linear approximation of SLO term i.

α_{il}, β_{il} specify the slope and y-intercept for piece l of the throughput-LLC linear approximation of SLO term i.

τ, φ specify the number of cores and LLC size available on each machine.

U_i is the maximum input load level that below which the latency objective of SLO term i is satisfied across all LLC allocations.

I_{ijk} indicates whether instance j of SLO term i is active on machine k.

N_k is set if and only if machine k is active – *i.e.,* at least one instance is assigned to it.

C_{ij} indicates the amount of LLC allocated to instance j of SLO term i. For an active instance, each such variable takes a value between the minimum and maximum permissible LLC allocation.

λ_{ijl} indicates whether linear fit l is chosen for instance j of SLO term i.

Below we briefly describe the goal of each constraint in the order they appear in Listing 1:

1. An instance runs on at most one machine.
2. The total LLC allocated to instances assigned to a machine is less than or equal to the machine's total LLC size (φ).
3. A machine is active when there is at least one instance running on that machine, and an active machine may host no more instances than its available cores (τ).
4. The aggregate throughput of instances of each SLO is greater than or equal to its target throughput (θ_i).
5. Linear piece l of a profile is chosen if and only if the LLC allocated to instance j of SLO term i lies in the range corresponding to piece l of the throughput-LLC linear approximation.
6. Exactly one linear piece is chosen when instance j of SLO term i is active, otherwise, none is chosen.

For simplicity, we assume a homogeneous infrastructure and that each SLO term instance requires a single CPU core; the MILP could be adjusted to account for differences if necessary. To account for small performance degradation despite ResQ's isolation (see §4.4), we include a 3% discount in U_i.