Microservice Benchmarking on Intel IPUs running Napatech Software

Zhuangzhuang Zhou[1], Yueying Li[1], Stuart Johnson[2], Nick Finamore[2], Charlie Ashton[3], Robert Cyphers[3], and Christina Delimitrou[4]
[1] Cornell University, [2] Intel Corporation, [3] Napatech, [4] MIT

Cloud computing services are increasingly moving from monolithic designs, to fine-grained, modular microservices [6, 8–10, 12, 13]. Microservices can be hosted and managed both by cloud operators, e.g., cloud services like Twitter, eBay, Amazon, Netflix, etc. all use microservices, or they can be designed and managed by end users, running on public or private cloud infrastructures.

Microservices are appealing for several reasons, including accelerating development and deployment, simplifying correctness debugging, facilitating elasticity, and enabling software heterogeneity [4, 7, 12].

At the same time microservices also introduce significant networking overheads, motivating the need for hardware offloads through the use of Infrastructure Processing Units (IPUs).

This white paper quantifies the performance characteristics of cloud microservices built with different programming frameworks on an Intel IPU running Open vSwitch (OVS) offload software from Napatech.

Open vSwitch (OVS)

Virtualized environments need the ability to forward traffic between Virtual Machines (VMs) or containers and the outside world.

Open vSwitch (OVS) is software used in virtualized environments to provide network connectivity between the outside networks and the VMs or Containers, or between the VMs running on the same host. But software-only implementations of OVS are CPU intensive and often unable to meet the performance and scale requirements for data center deployments of virtualized applications.

Intel^{TM} C5010X IPU with Napatech Link-Virtualization™ Software

The Intel IPU C5010X with Napatech Link-Virtualization software is an FPGA-based OVS hardware offload solution based on Data Plane Development Kit (DPDK). The use of hardware offload results in a significant reduction of CPU cycles required to move network data which results in much lower overall Total Cost of Ownership (TCO).

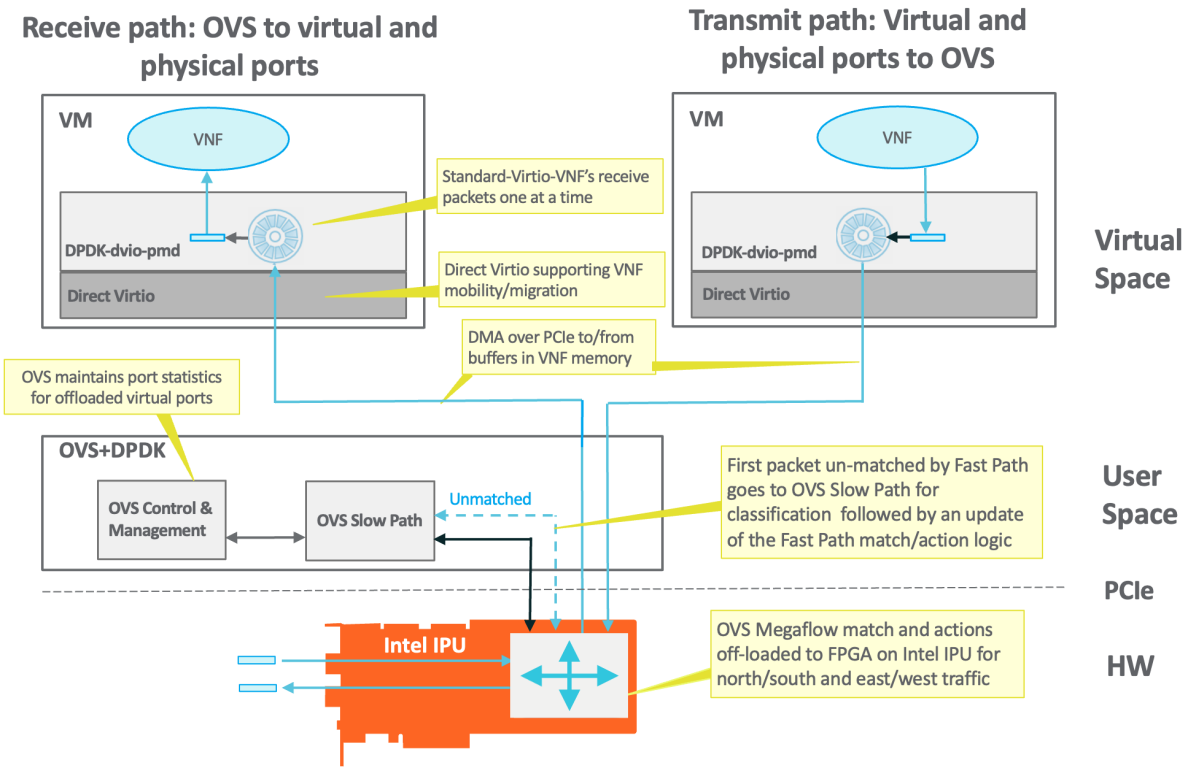

Figure 1: Packet walkthrough with OVS hardware offload.

How Does It Work?

The first packet of any new flow will fail to match a flow-table lookup in the IPU and must be forwarded to the OVS running on the host for classification (Slow-path processing). Once OVS determines the proper handling of the new flow, a flow table entry will be added in both the OVS software data path and the FPGA data path. The two flow tables are kept always synchronized by the driver software running in the hypervisor. Once the flow handling rules are programmed, the initial packet is sent back to the IPU for normal processing.

All subsequent packets from the same flow will be handled by the IPU FPGA (Fast-path) logic according to the prescribed match/action rule.

Slow-path packet handling consumes host CPU cycles and requires copying data between the IPU, the host memory-space, and the VM/VNF (Virtualized Network Function).

Fast-path packet handling bypasses sending data to the OVS software running in the host: instead of transferring data through OVS, the IPU transfers data directly to the memory of the VNF using Direct Memory Access (DMA) over the virtio data-path. Figure 1 shows the lifetime of a packet when the IPU is enabled.

Methodology

Table 1 details the specification of the IPU-enabled servers.

Table 2 details the specification of the Intel IPU.

Table 3 details the specification of the Napatech OVS offload software.

We evaluate the performance of the hardware offload on three classes of multi-tier low-latency applications, built with different frameworks and communication models. These applications can either be managed by cloud operators, as is the case with Amazon or Twitter's microservice graph, or designed and managed by end users running on public or private environments. We identify three applications that vary in the complexity of their topologies and inter-service dependencies,

Server Spec (×2)	Inspur Dual Socket
Processors	Ice Lake Gold 6338
Cores per Processor	32
Core Freq. (base/turbo)	2.0GHz/3.2GHz
Memory	512GB
Boot Drive	480GB
NVMe Storage	2x or 4x 1.6TB P4610
Lab Network NIC	10Gbe Fortville
IPU	Silicom C5010X
OVS Offload Software	Napatech Link-Virtualization 4.4

Table 1: Server cluster setup used for benchmarking.

IPU Spec	C5010X
FPGA	Intel Stratix 10 DX 1100
SoC	Intel Xeon D-1612
Network Interface	Dual 25 GbE (2 x 10/25 Gbps)
Supported Pluggable Modules	SFP+ / SFP28
Host Interface	PCIe Gen3 x8 (x16 physical)
On-Board Memory	4GB DDR (FPGA) + 16GB DDR ECC (SoC)
Management	PXE and SATA SoC Boot Options
Size	Full Height, Half Length (111.15mm x 167.65mm)
Power	75W Max with Passive Cooling

Table 2: Intel IPU specifications.

in the amount of CPU, memory, and network resources they require, and in the frameworks they leverage for individual service tiers to interact with each other.

- First, we evaluate a three-tier application with NGINX as the front-end webserver, memcached as the in-memory caching tier, and MongoDB as the back-end database in persistent storage. Fig. 2 shows the architecture of this application. Clients interact with the front-end webserver (NGINX), which first checks if a key-value pair is cached in memcached. If so, it returns the value to the client. If not, NGINX interfaces with MongoDB to fetch the output and additionally cache it in memcached.

- Second, we evaluate a five-tier chain topology built with Dapr's [2] pub-sub communication model that uses message passing for data transfers across tiers. Fig. 3 shows the application's topology. Each tier performs CPU-intensive computation for a user-specified amount of time, before broadcasting its output to the downstream tier.

- Third, we characterize the social network application from DeathStarBench [1, 12], an RPC-based, multi-tier service that implements a broadcast-style social network, where users can create posts, follow other users, and browse their timelines. The social network is designed in the style of Twitter, reuses widely deployed cloud services, like NGINX, Memcached, and MongoDB, and is driven by an input load that resembles the traffic pattern and user graph characteristics of a scaled down version of Twitter. Fig. 4 shows the topology of the social network. All microservices are deployed with either Docker Swarm or Kubernetes

OVS Offload Software Spec	Napatech Link-Virtualization 4.4
Switching Performance (64B packets)	300Mpps Port-Port / 65Mpps Port-VM-Port
VirtIO Support	Fully-Accelerated VirtIO 1.1 with vDPA
Live Migration Support	VirtIO 1.1
Host OS Support	Red Hat RHEL 8 / Ubuntu Server LTS on Request
OpenStack Support	Train + Victoria
DPDK Support	20.11
OVS Support	2.15
Linux NetDev Support	Kernel 5.0+
Other Networking Features	V(X)LAN / Q-in-Q / Link Aggregation / QoS

Table 3: Napatech OVS offload software specifications.

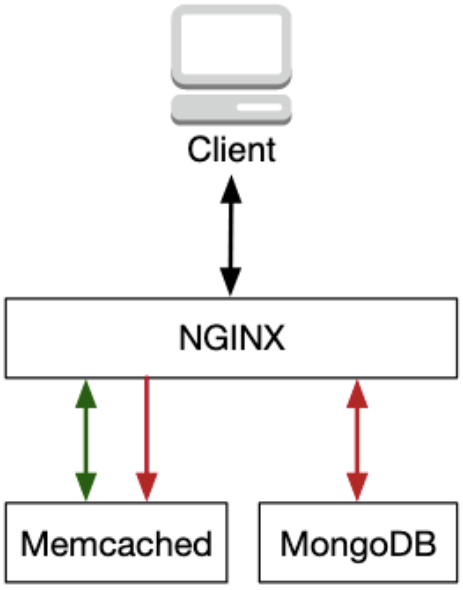

Figure 2: Microservice topology for the 3-tier NGINX-memcached-MongoDB application.

in single-concerned Docker containers, and communicate using RPC requests (unless otherwise specified), via the Apache Thrift frameworks.

Users (clients) send requests, which first reach a load balancer, implemented with NGINX. Once a specific webserver is selected, also in NGINX, the latter uses a php-fpm module to talk to the microservices responsible for composing and displaying posts, as well as microservices for advertisements, search engines, etc. All messages downstream of php-fpm are using Apache Thrift RPCs [5].

Users can create posts embedded with text, media, links, and tags to other users. Their posts are then broadcast to all their followers. Users can also read, favorite, and repost posts, as well as reply publicly, or send a direct message to another user. The application also includes machine learning plugins, such as ads and user recommender engines, a search service using Xapian, and microservices to record and display user statistics, e.g., number of followers, and to allow users to follow, unfollow, or block other accounts. The service's backend uses

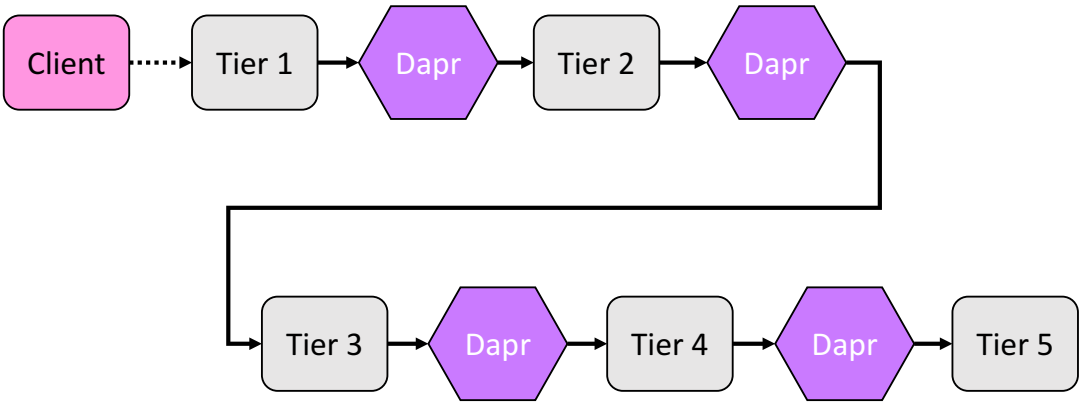

Figure 3: Microservice topology for 5-tier chain pub-sub service built with Dapr.

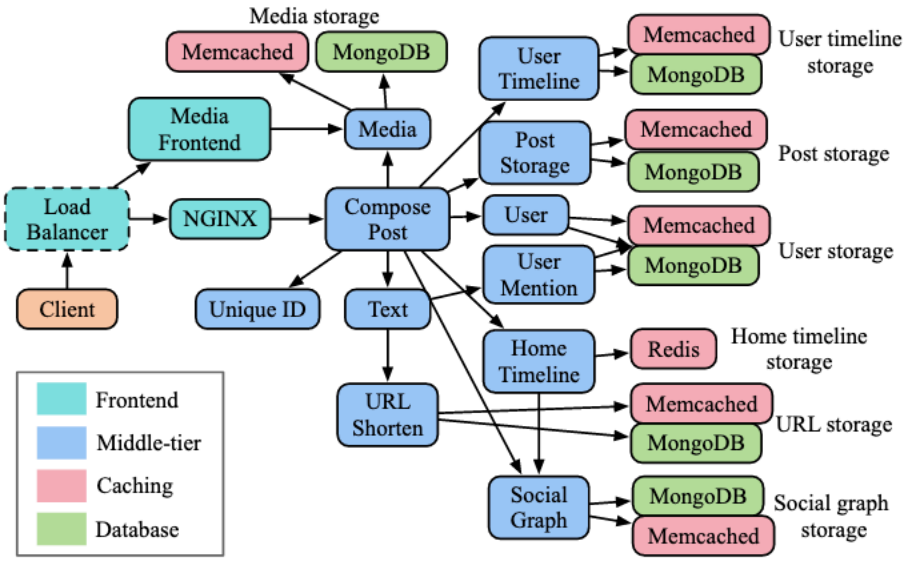

Figure 4: Microservice topologies for Social Network.

memcached for caching, and MongoDB for persistent storage for posts, profiles, media, and recommendations.

Finally, the service is instrumented with a distributed tracing system based on Jaeger [3], which records the latency of each network request and per-microservice processing; traces are recorded in a centralized database. We additionally record the utilization of different shared resources per microservice. The application has been used extensively for studies of hardware acceleration for networking [14], performance debugging in microservices [11], and QoS-aware cluster management [15].

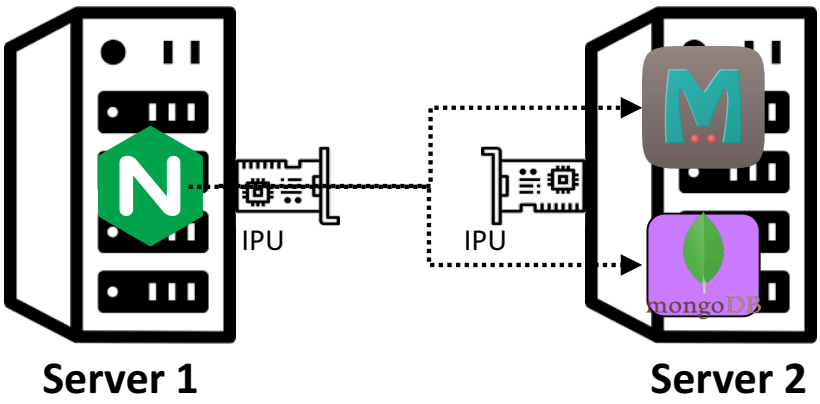

Figure 5: Placement of application tiers on physical servers for the 3-tier service.

Experimental Results

Three-tier TCP-Based Application

Fig. 5 shows the physical placement of each service in the 3-tier application across the two OVS-enabled servers. We ensure that dependent services are placed on different physical machines, such that all traffic goes over the hardware offload (Intel IPU), when it is enabled.

Fig. 6 shows the load-latency curves for the 3-tier application, as input load increases when the offload is disabled or enabled. In both cases we show average and tail (99^{th} percentile) latency.

At low loads, the difference in latency is low, as neither the standard NIC nor the IPU get saturated. For higher loads, there is a significant difference in latency depending on whether the offload is enabled or not. When the hardware offload is disabled, the application saturates at 26kQPS (26,000 queries per second), after which point a large fraction of requests get dropped. When the offload is enabled, the service can reach 28kQPS without a significant increase in latency, and only starts dropping requests after 32kQPS. This means that the same hardware can accommodate 31% more requests without a degradation in user experience, increasing the infrastructure's resource efficiency. Alternatively, for high loads, e.g., 22kQPS, enabling the offload achieves 8× lower latency compared to disabling it.

Fig. 7 (left) shows the CPU utilization for each service in the 3-tier application with/without the hardware offload, as input load increases. In both cases, NGINX consumes the highest amount of resources, however, when the offload is enabled that number is reduced by 8%-30%; 13% on average. The results are similar for the two databases as well, although the reduction is less pronounced, since neither of them are CPU-bound to begin with.

Finally, Fig. 7 (right) shows the amount of network bandwidth consumed with/without the hardware offload. For the default packet size the application uses, both configurations achieve similar network bandwidth usage, as most key-value pairs are small. At high loads (over 26kQPS), when the offload is disabled network bandwidth approximates saturation. To push bandwidth usage further when the offload is enabled, we alter the default request distribution to transfer larger objects (roughly double the size), which achieves double the network bandwidth usage. Object size cannot be increased further due to CPU saturation on the host servers.

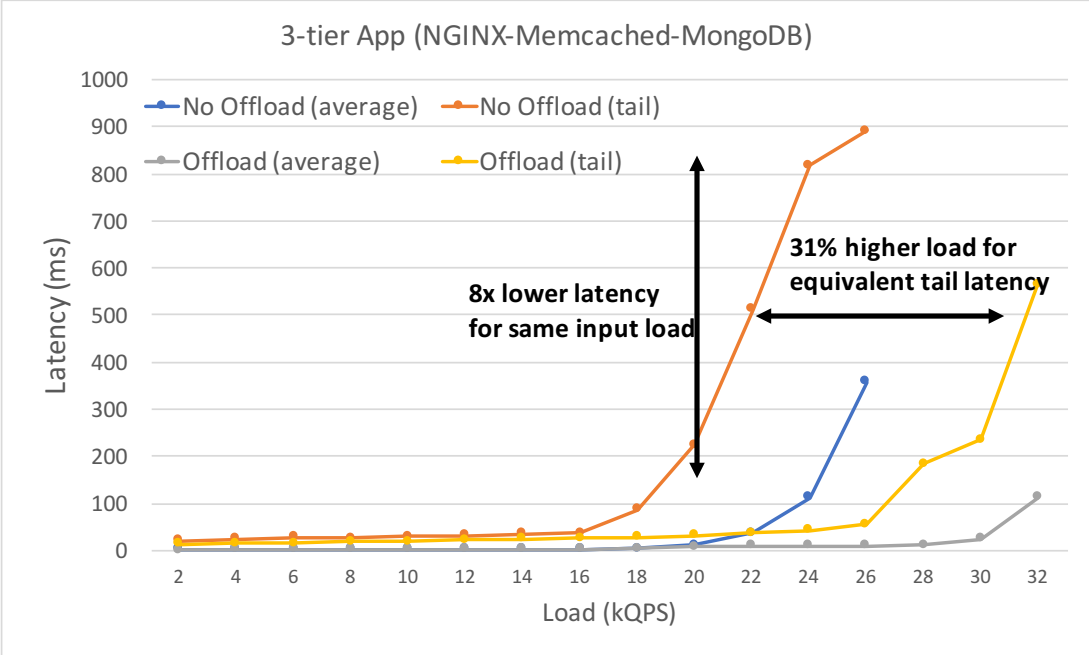

Figure 6: Mean and tail latency across loads when OVS is enabled and disabled.

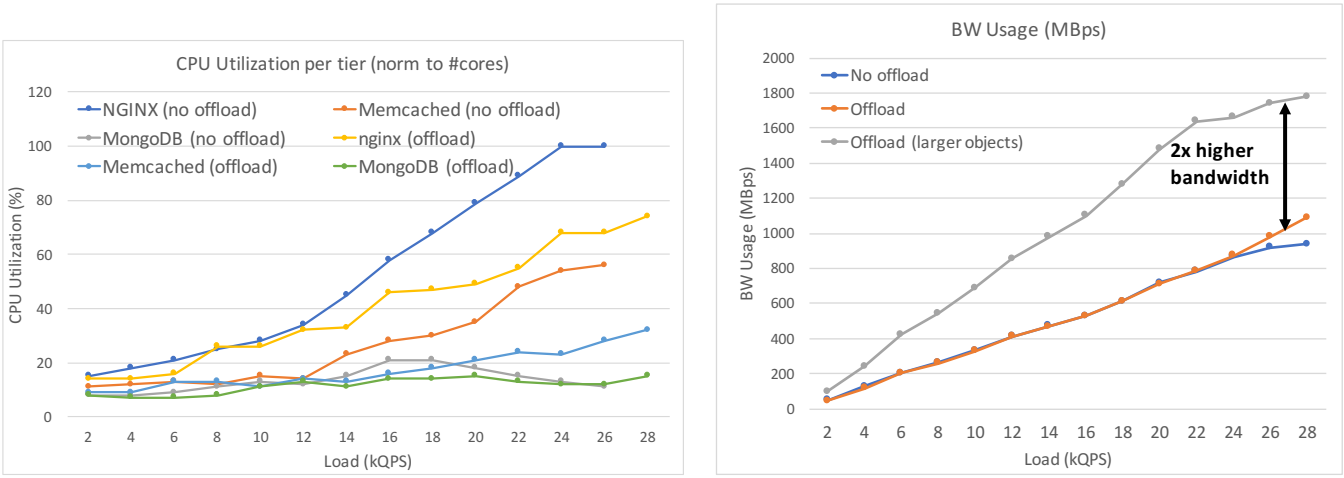

Figure 7: Average CPU utilization per tier with and without the offload (left), and average network bandwidth usage with and without offload (right). The last line shows bandwidth usage with larger objects to saturate the IPU's bandwidth.

Five-tier Pub-Sub Application

Fig. 8 shows the physical placement of each service in the 5-tier pub-sub application across the two OVS-enabled servers. We ensure that dependent services are placed on different physical machines, such that all traffic goes over the hardware offload, when it is enabled.

Fig. 9 shows the load-latency curves for the 5-tier application, as input load increases when the offload is disabled or enabled. In both cases we show average and tail (99^{th} percentile) latency. At low loads the difference in latency is modest, but it increases substantially for higher loads. When offload is disabled, the application saturates at 120kQPS, beyond which point requests start dropping. When offload is on, the application saturates at 145kQPS and allows 32% higher load without an increase in average or tail latency. This is beneficial for cloud providers, as they can accommodate more work on the same hardware. This is especially important with microservices,

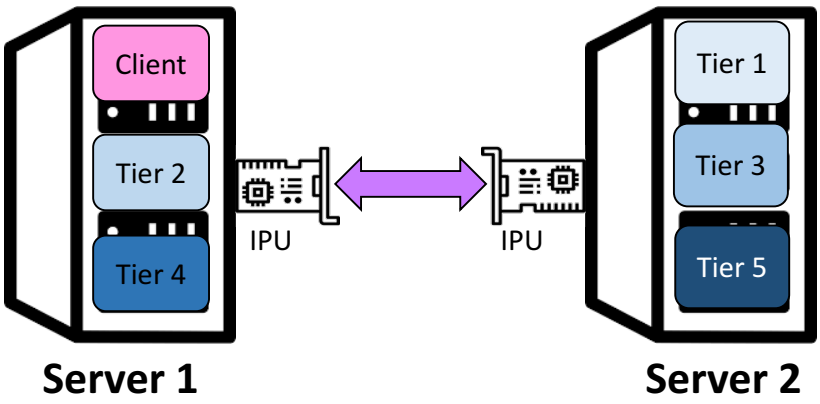

Figure 8: Placement of application tiers on physical servers for the 5-tier pub-sub service.

as the degree of multi-tenancy is much higher compared to traditional monolithic application design.

Fig. 10 (left) shows the difference in average CPU utilization across the 5 tiers with/without the hardware offload, across input loads. At low loads the difference is small as neither system is saturating the available cores. For higher loads (over 60kQPS) the difference becomes more substantial, and continues to increase as input load goes up. The average difference in CPU utilization between the two configurations is 19.8%. By offloading some of the network processing to hardware, the host CPU is able to accommodate more requests, achieving better overall resource efficiency.

Finally, Fig. 10 (right) shows the difference in end-to-end throughput with/without the hardware offload. For the default request size distribution, which primarily uses small objects, the difference in throughput is small, as neither system is saturated. The configuration without the hardware offload approaches saturation for loads beyond 130kQPS, and increasing the object sizes quickly saturates the standard NIC. In contrast, when the offload is enabled, we are able to increase the object size and overall throughput by 2.3× on average before the host CPUs become saturated. While this still does not saturate the IPU's bandwidth capabilities, it accommodates 2.3× more data transfers per second, enabling more data intensive low-latency computation, such as real-time inference and serverless analytics.

RPC-Based Microservice Topology

We now evaluate the network offload on an end-to-end microservice topology implementing the RPC-base social network. Fig. 11 shows the physical placement of each microservice in the end-to-end topology across the two server nodes. Microservices are placed, such that all traffic goes over the hardware offload, when it is enabled.

Fig. 12 shows the benefit of enabling the hardware offload across latency percentiles for the end-to-end Social Network application. While there is some benefit at high percentiles, the gains are limited owing to the application primarily transferring small packets, where host CPU is the constraining factor, and the fact that for larger transfers, the overhead of serialization/deserialization dominates RPC processing latency. In the discussion section we briefly discuss ways to optimize small RPC execution in the hardware offload. The difference in CPU usage and network throughput are similarly low. When the offload is enabled, the server operates at 29.4% average CPU utilization and reaches 2.1GB/s throughput.

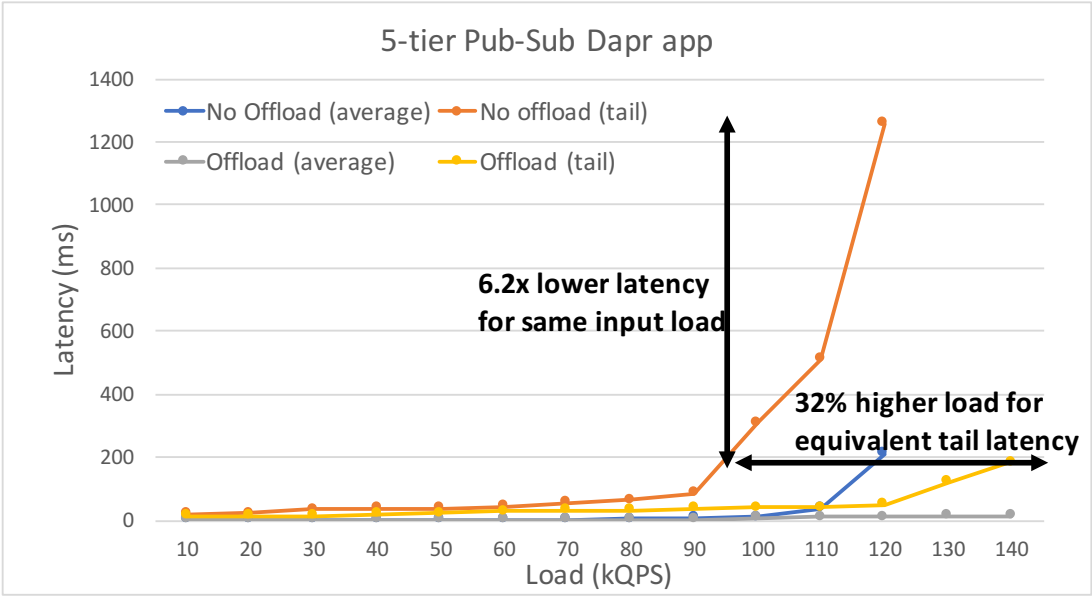

Figure 9: Mean and tail latency across loads when OVS is enabled and disabled for the 5-tier pub-sub microservice topology.

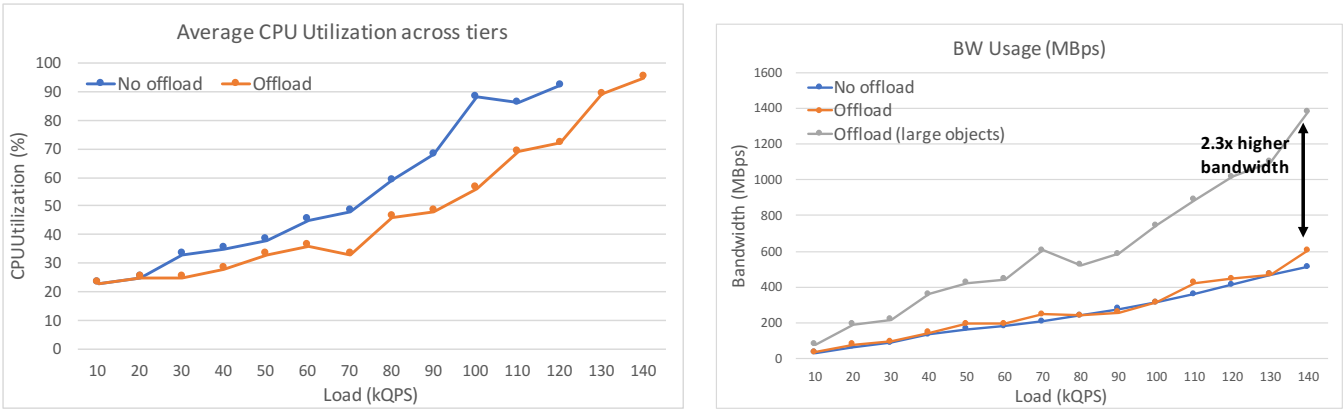

Figure 10: Average CPU utilization with and without the offload (left), and average network bandwidth usage with and without offload (right) for the 5-tier pub-sub topology. The last line shows bandwidth usage with larger objects to saturate the IPU's bandwidth.

Discussion

As shown in the evaluation above, the hardware offload offers significant performance and utilization benefits for low-latency microservices. These translate to both better latency for the end user, for the same input load, as well as higher resource efficiency for the cloud operator, given that more work can be accommodated on the same resources, without an increase in observed latency.

Out of the three examined development strategies for microservices, the pub-sub model displayed the highest gains, followed by the TCP-based 3-tier application. The RPC-based topology experienced modest performance and utilization gains for high latency percentiles.

Given that RPCs remain a popular communication framework for low-latency microservices, offloading more of the RPC processing stack to the IPU, including serialization-deserialization, which dominates CPU usage for large packets, as well as optimizing for small data transfers,

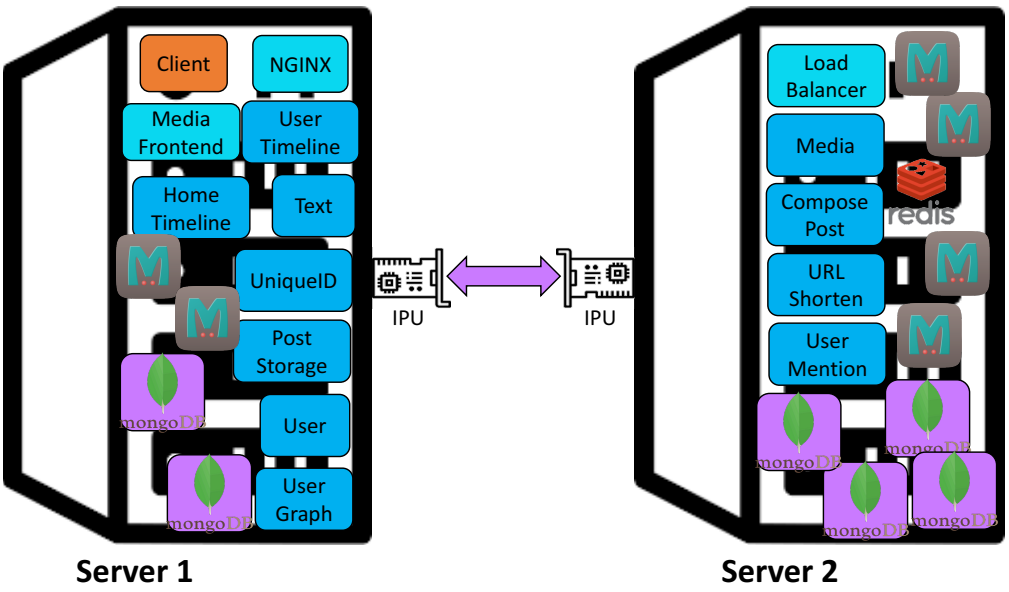

Figure 11: Placement of application tiers on physical servers for the Social Network topology.

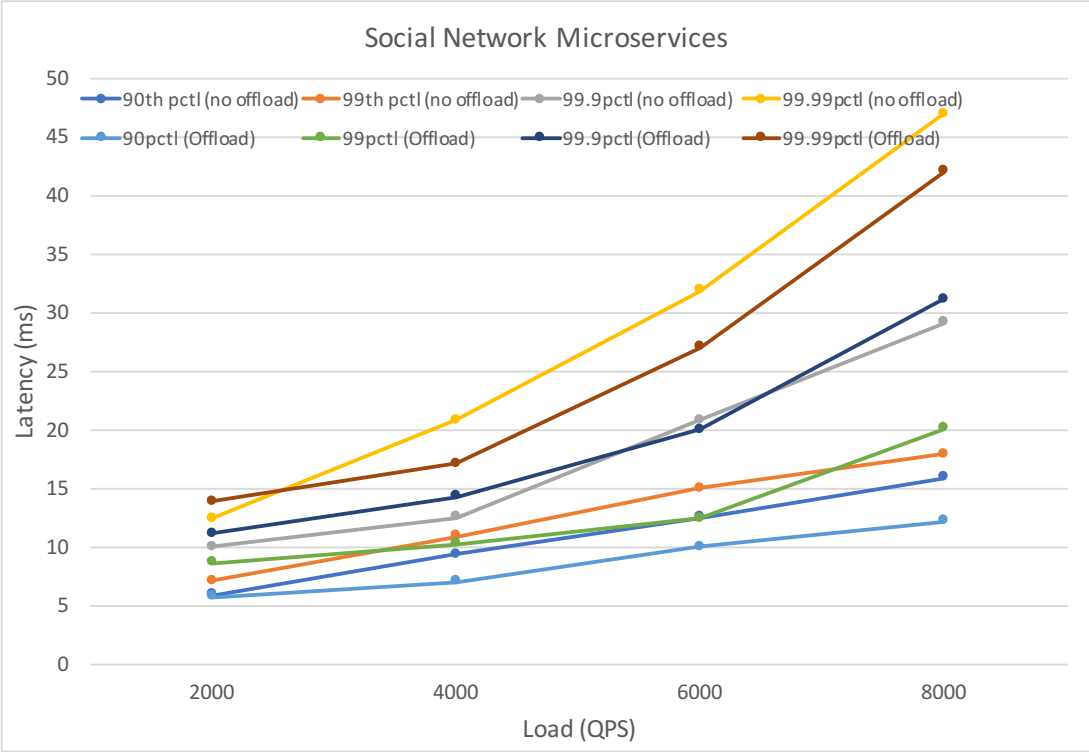

Figure 12: Different latency percentiles for the social network topology, as input load increases with and without the Intel IPU enabled.

which are more prevalent in low-latency microservices, can enhance these performance gains in the future.

References

[1] DeathStarBench: Open-source benchmark suite for cloud microservices. https://github.com/delimitrou/DeathStarBench.

[2] Distributed Application Runtime. https://dapr.io/.

[3] Jaeger: Open-Source, End-to-End Distributed Tracing. https://www.jaegertracing.io/.

[4] The Evolution of Microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[5] Apache Thrift. https://thrift.apache.org, 2017.

[6] L. Barroso and U. Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers, 2009.

[7] A. Cockroft. Microservices: Why, what, and how to get there. In *Microservices Workshop All Topics Deck*, 2016.

[8] J. Dean and L. A. Barroso. The Tail at Scale. In *CACM, Vol. 56 No. 2*.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[10] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proc. of ASPLOS*. Salt Lake City, 2014.

[11] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[12] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[13] Y. Gan, Y. Zhang, K. Hu, Y. He, D. Cheng, and C. Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[14] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

[15] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2021.

Acronyms

DDR Double Data Rate (Synchronous Dynamic Random-Access Memory)

DMA Direct Memory Access

DPDK Data Plane Development Kit

ECC Error correction code (memory)

FPGA Field Programmable Gate Array

IPU Infrastructure Processing Unit

OVS Open vSwitch

PCIe PCI Express (Peripheral Component Interconnect Express)

PXE Preboot eXecution Environment

QoS Quality of Service

RPC Remote Procedure Call

SATA Serial AT Attachment

SFP Small Form-factor Pluggable

SoC System on a chip

TCO Total Cost of Ownership

vDPA Virtual data path acceleration

VirtIO virtual input and output

VM Virtual Machine

VNF Virtualized Network Function