Solution Brief

Microservices Solution - Optimizations with Intel® Xeon® Scalable Processor

Accelerate Microservice Networking Performance with 4th Gen Intel® Xeon® Scalable Processor

Authors

Mrittika Ganguli
Kelley Mullick
John O'Loughlin
Xiaobing Qian
Ismo Puustinen
Renzhi Jiang
Andrei Palade
Amruta Mishra

Executive Summary

Microservice architectures are decoupling applications from maintaining and managing infrastructure operations to performing only the required business logic. Infrastructure tasks such as transferring data, performing health checks, and rate limiting are being decoupled from orchestrators like Kubernetes (K8s) and are being taken up by service mesh using sidecar proxies, forming a software-defined data plane for microservices.

Applications running as microservices still perform tasks for which logic has to be replicated by each microservice. Service mesh decouples these operations. A service mesh deployment makes use of iptables to establish network connections between pods and nodes, managing the networking and port forwarding rules.

Each layer of networking adds 15-30% to the overall latency and reduces the total number of requests per second (RPS). We present the characterized and optimized performance of Kubernetes CNIs like Calico, web proxies like NGINX, service mesh like Istio-Envoy, and remote procedure call frameworks for gRPC. We recommend using eBPF, DPDK, and VPP-based optimized networking stack, which provide more than 2x performance with Intel® Xeon® processor accelerator offloads and Xeon processor-specific instructions (ISAs).

This document is part of the Network and Edge Platform Experience Kits.

Introduction

Traditionally, enterprise applications were created as monolithic applications, meaning the applications were written and integrated into a single software image. These applications contained all the code and functionality to perform ALL the business activities the application could perform. The problem with the single image approach is that 1) scalability requirements often vary dramatically across the code, and 2) it is often desirable to use different programming languages that run on different node types depending on function. These monolithic applications became large with increasing complexity and requirements, and ultimately negatively impact businesses as well as the ability to scale.

To address this issue, applications were rearchitected as "Microservices". Microservices design breaks applications into smaller, independent, and self-contained functions that are managed and maintained separately. This is typically done in a container-based environment such as Kubernetes. Having separate functions means that scale is achieved by expanding the number of containers to perform a task. Microservices use remote procedure calls (RPCs) to invoke libraries on remote systems. The RPC stacks are computationally costly, have high latency, and expose the program to higher latency variability (tail latency) across the network rather than within the node.

Key to any microservice deployment is the ability to enforce service-level agreements for each node running microservices as well as the overall service. The predictability of an application's response time is measured in terms of jitter. Jitter refers to the variability of latency, the slight variation (earlier or later) in turnaround time when a response is received, rather than the average latency in a server. The distribution of jitter increases with scale. The tail latency, i.e., the largest outliers within the set of system response times, in an otherwise fast system, becomes an increasingly significant factor for consideration as the data center scales with more servers. Tail latency has a visible impact to the client but it is hard to isolate exact sources within a data center. Thus, any reduction in tail latency is important for achieving high utilization of data center resources.

The second challenge in microservices-based workloads is how to scale the application with the optimized utilization of the underlying compute resources. Often the system contains a large number of different microservices that have complex interactions. When unpredictable workloads arrive, it is problematic to identify the scaling needed and evaluate the amount of data center resources needed.

In addition, a service mesh consisting of proxy and security services is often needed as a side car container. The RPC, service mesh, transport, and container network interface amounts to a very costly network stack, impacting overall performance and the ability to scale. Furthermore, Kubernetes allows for automation of many management tasks such as provisioning and scaling but impacts performance. This results in four top challenges for efficient deployment:

1. Reduction of tail latency
2. Application scalability /utilization
3. Throughput
4. Security

As data volume grows and workload requirements shift to address new business opportunities, new performance bottlenecks arise. Thus, performance of the system can be impacted and the third challenge on throughput emerges. Throughput, or the amount of data that can be transferred from one location to another in a given amount of time, is measured in transactions per second. Improving throughput in a microservices application is a key metric to be considered at scale.

Finally, some of the inherent benefits from microservices architecture pose new security risks when deployed at scale. For example, given that the microservice containers are smaller and widely distributed, the chance of propagating a threat increases when changes are made. Increasing the number of infrastructure layers that require protection also adds increased risk. Attacks can come through weaknesses in the application code, infrastructure layers, or networking within the data center. Having compute resources accessible on the public cloud also increases the risk of cyberattacks. Protecting the data center resources and underlying infrastructure through robust security policies can help mitigate the security risks.

Addressing the Challenges

Let's take a detailed look at where performance bottlenecks exist, and the steps data center operators can take to more efficiently deploy microservices. Figure 1 depicts a typical microservices design for an application and network flows within the services. Service request communications happen with gRPC (remote procedure call) calls to specific applications. The applications perform different functions such as web requests, data transfer, and compute, requiring time to perform the task, which is measured in transactions per second. The latencies result from the time it takes to process requests, the communication within the microservice, and how the response is communicated outside the request. The direct correlation is between the queries per second resolved by the applications in pods and the tail latencies of each of these resolved queries. Unlike Layer 3 (IP) benchmarking that considers packet delay variation or jitter or round-trip latencies, Layer 7 (HTTP, HTTPS, gRPC, and so on) latency benchmarking considers latencies for the time required for a certain percentile of queries to a particular application to be completed. For example, the P90 latency represents the highest latency value with respect to the lower 90 percent of all recorded transaction latencies.

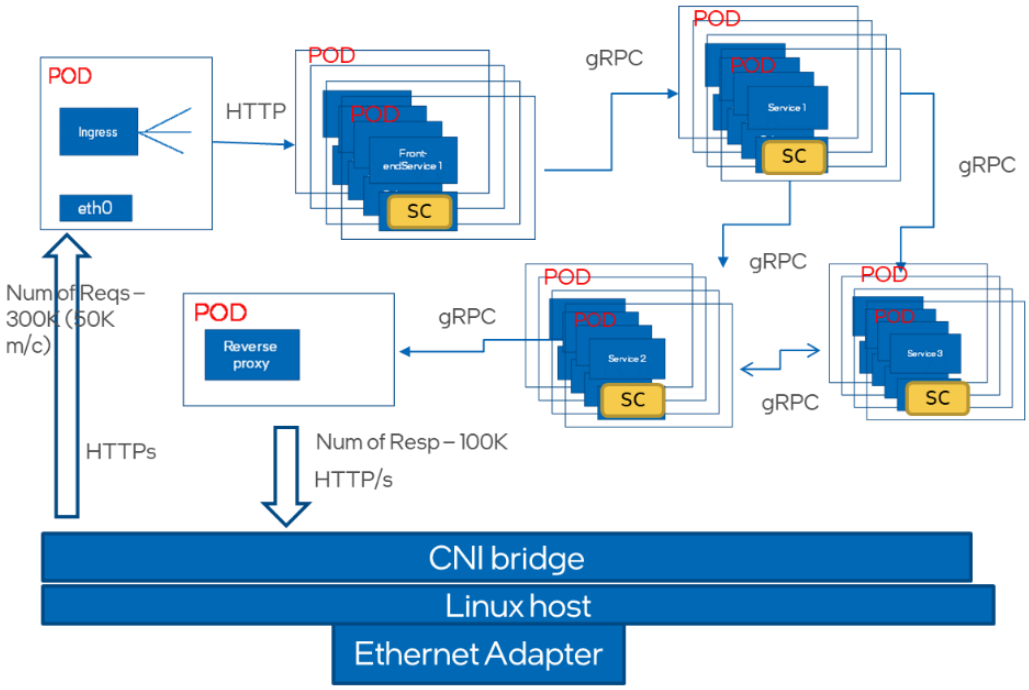

Figure 1. Industry Trends for Microservices Deployment

Performance bottlenecks in the context of microservices workloads are depicted in Figure 2.

(1) the overhead of the Linux Transmission Control Protocol/Internet Protocol (TCP/IP) stack and Kubernetes networking

(2) Higher core context switches; Higher inter-core transfers and caches misses; higher inter-NUMA domain challenges; Higher security and performance challenges

(3) Exploding E-W traffic; New security requirements; Increasing use of service mesh; Multiple traversals across the TCP/IP stack

(4) Higher inter-node communication and overhead due to tunnels. The adoption of CNI and service mesh introduce large latencies

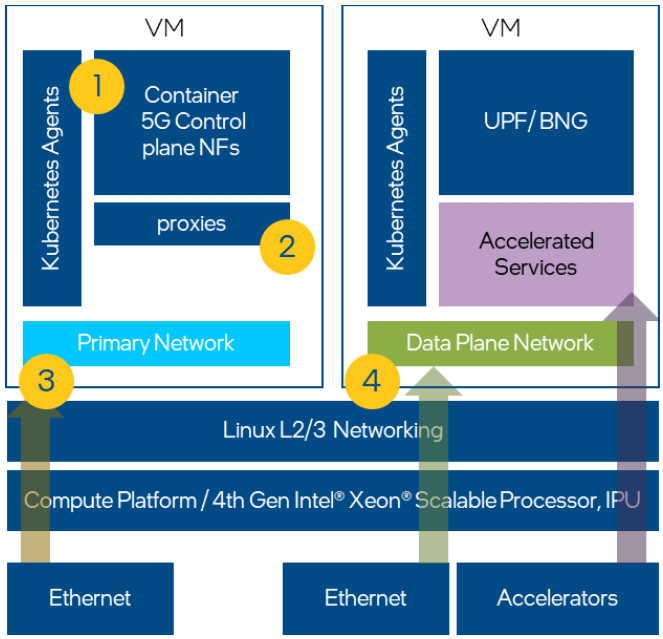

Figure 2. Networking Layer Performance Bottlenecks

Performance Advantage of Intel Performance Characterization

Kubernetes, a de-facto software and toolset for orchestration and management at the edge can customize its functionality similar to building blocks and enable application developers to develop and deploy edge services using a microservices approach with various container network interfaces (CNI) like Calico. Thus, we focus on characterizing the performance of baseline Linux, Calico CNI, Envoy, NGINX, and DSB.

As summarized in Table 1, each bottleneck is addressed.

Table 1. Addressing Networking Layer Performance Bottlenecks

Bottleneck	Software Stack	Hardware Acceleration
L2/3 TCP performance with large packet copies	Bypass kernel eBPF or user mode stack with VPP and DPDK in Calico	Intel® Data Streaming Accelerator (Intel® DSA)
L3 routing	Bypass kernel eBPF or user mode stack with VPP Calico	
HTTP1 &2 performance - Core scheduling and load balancing	Envoy or NGINX with core load balancing offload	Intel® Dynamic Load Balancer (Intel® DLB)
HTTPs performance with TLS and crypto	NGINX and Envoy with OpenSSL and BoringSSL crypto offload	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) and Intel® QuickAssist Technology (Intel® QAT)

Layer 2/3 Performance with Calico

Cloud deployment desperately needs a Layer 2/3 networking solution. Today scaling is hindered by Linux kernel network stack. Calico and Cilium –are two Kubernetes (K8s) bridge (CNI) based on kernel stack. Kernel bypass using eBPF is available in both Cilium and Calico.

Alternative CNIs are Calico and Cilium eBPF provide near bare metal performance. Calico is a flexible, open source CNI that supports multiple data planes such as standard Linux, eBPF, VPP, and Windows. Calico provides both network and IPAM plugins to Kubernetes. In other words, as a CNI, Calico needs to create the networking between pods and assign IPs to pods. Calico can also provide restricted access to pods. Calico by default uses the host/nodes kernel routing tables to route traffic between the pods. Tunnels are used to connect each node to every other node in the cluster. The routing rules are synced and updated using BGP across the cluster.

Vector Packet Processor (VPP)

VPP is a highly optimized layer 2 - layer 4 network stack. It has a pluggable graph-based structure that allows it to perform functions such as load balancing, switching, and routing. It supports multiple interface types such as tap tun, which interfaces to the kernel, and memif, which is an optimized user space interface.

Calico VPP

Calico can switch out its data plane to use VPP. Calico VPP takes over the Ethernet Adapter and creates a tap tun interface to the host that uses the name that was originally assigned to the Ethernet Adapter. As the new tap tun interface to VPP has the same name as that was assigned to the physical Ethernet Adapter, all the host routing rules are kept intact. A tap tun interface is also created and used for each pod. VPP does the routing before the traffic hits the host OS. Traffic for the pods is routed through the VPP routing table and does not need to interact with the host's kernel routing table. Although the routing is taken away from the kernel, packets still need to go through the kernel tap tun interfaces to be received by the pod.

Calico VPP with memif

Memif is a highly optimized software packet interface that includes critical features such as multi-queue and one-way zero copy.

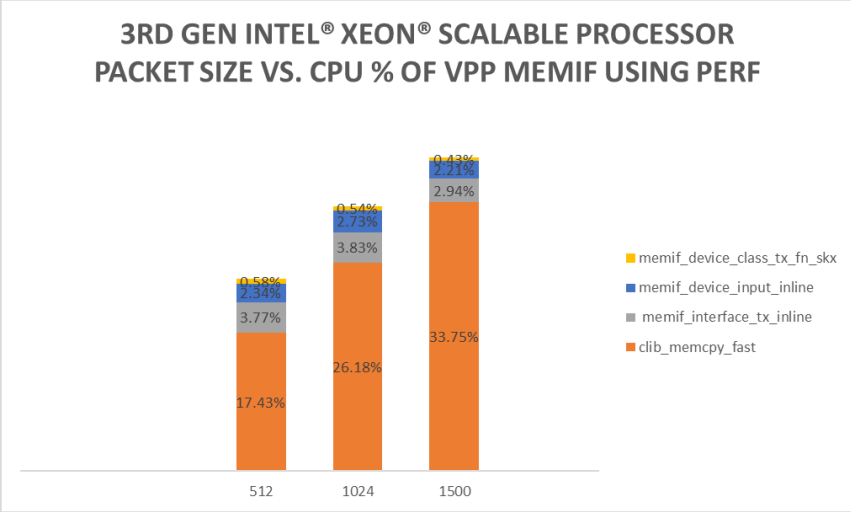

Figure 3. Calico VPP memif % of Cycles doing memcpys

Calico VPP provides memif user space packet interfaces to the K8s pods. Memif can be enabled through pod annotation and the pod interface is selected as memif or tun for ingress traffic based on ports. Memif boosts user plane applications as it completely bypasses the kernel. By default, VPP is the controller for the memif interface. Figure 4 is the Calico VPP block graph.

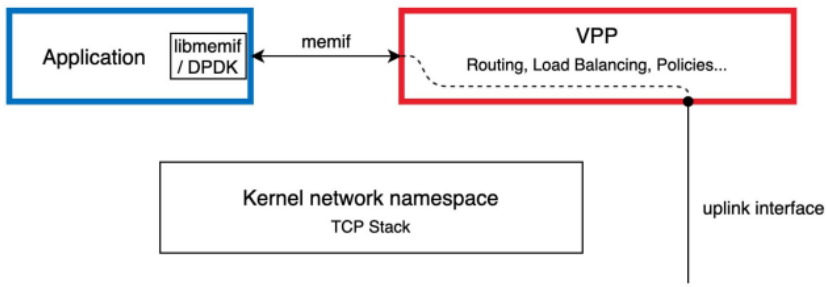

Figure 4. Calico VPP memif Block Graph

CPU cycle analysis shows 26–33% of the time is spent in memcpy operations in the memif interface (Figure 3). Focusing on accelerating the memcpy performance resulted in offloading the operation to a CPU accelerator when packet sizes were larger than cycles used in offload operations. Intel Data Stream Accelerator (Intel DSA) provides DMA offload as a high-performance data copy and transformation engine, starting with 4th Gen Intel® Xeon® Scalable processors. The platform exposes the hardware as PCIe devices. The integration of Intel DSA technology into VPP memif breaks through the CPU bottleneck for software data copy operations and improves the performance of Calico VPP memif interface, as shown in Figure 5.

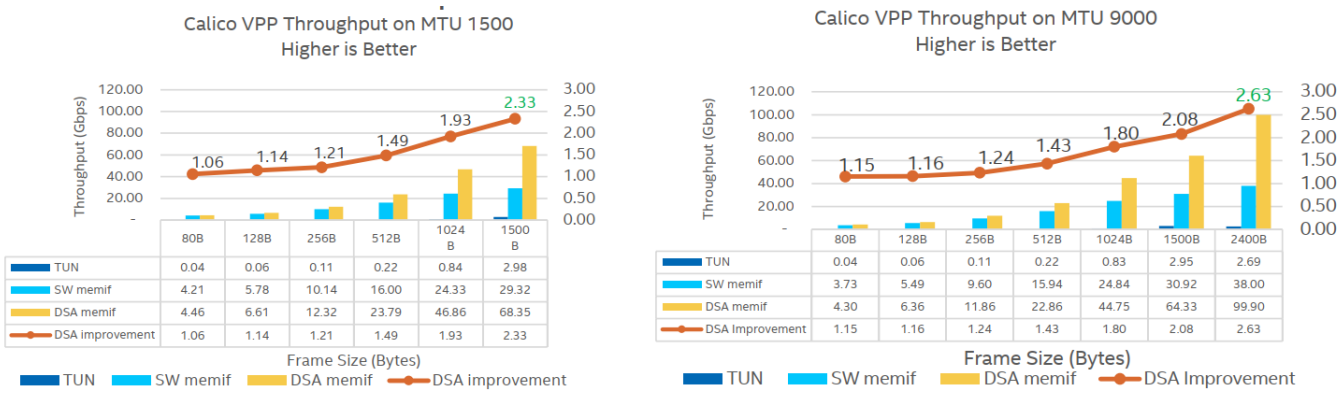

Figure 5. Calico VPP Frame Scaling Throughput with 1500 B and 9000 B MTU

In summary, we conclude the following benefits:

- Up to **2.33x** [MTU 1500, 1500 B] and **2.63x** [MTU 9000, 2400 B] higher single core throughput on 4th Gen Intel Xeon Scalable processor with Intel DSA memory copy compared with software memory copy
- MTU 9000 has better performance for large packet sizes
- Tun interface has low throughput across all frame sizes
- Calico VPP CPU usage is 100%

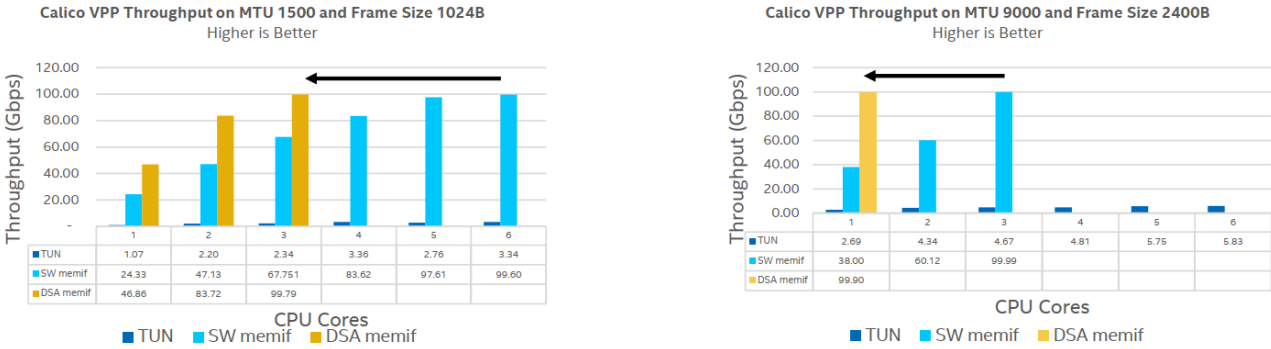

Figure 6. Calico VPP Core Scaling Throughput for 1500 B and 9000 B MTU

Figure 6 shows additional core saving benefits, as follows:

- Save **up to 3 cores** to achieve 100 Gbps throughput with 4th Gen Intel Xeon Scalable processor and Intel DSA with Intel DSA memif vs. software memif at MTU 1500 and 1024 B frame size
- Save **up to 2 cores** to achieve 100 Gbps throughput with 4th Gen Intel Xeon Scalable processor and Intel DSA with Intel DSA memif vs. software memif at MTU 9000 and 2400 B frame size
- Tun interface has low throughput across CPU cores
- Calico VPP CPU usage is 100%

eBPF-Based Bridge Using Calico

eBPF is used to bypass iptables routing in Calico. A baseline comparison shows that an 8 core (8C) allocation of 4th Gen Intel Xeon Scalable processor can be used for a maximum of 93 Gbps network bandwidth throughput (Figure 7) as opposed to a 3rd Gen Intel Xeon Scalable processor, which provides about 89 Gbps throughput. In comparison, a Calico VPP provides a 100 Gbps throughput using 6 cores.

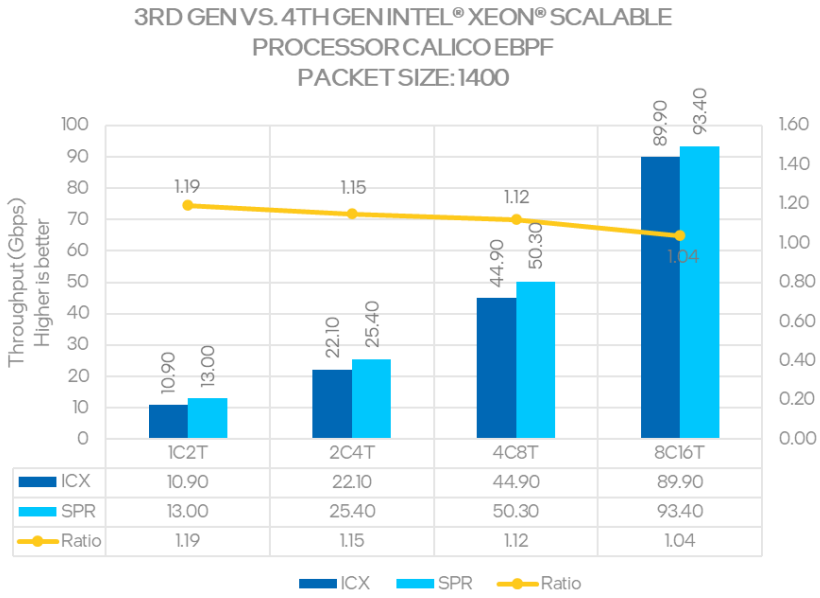

Figure 7. eBPF Benefits in Calico in 4th Gen Intel Xeon Scalable Processors[1]

[1] In this figure, ICX represents 3rd Gen Intel® Xeon® Scalable processor and SPR represents 4th Gen Intel® Xeon® Scalable processor

See backup for workloads and configurations. Results may vary.

NGINX

In microservice environments, web servers and proxies like NGINX distribute requests through worker threads and processes. The request distribution across cores may not be even, especially if the requests are of different object types and sizes. Some cores are more occupied than the others. NGINX has a software load balancer but takes many CPU cycles and does not do object prioritization. The connection load balancing in such environments depends on kernel scheduling.

NGINX with Intel Dynamic Load Balancer

The NGINX HTTP2/3 distributor was enhanced using Intel® Dynamic Load Balancer (Intel® DLB) to offload the distribution of requests across worker cores at the server. The NGINX thread pool is designed to eliminate the blocking issue, especially in environments with heavy I/O. It offloads the tasks into a thread pool and when one task is blocked by the system, the other tasks can be picked by the free thread immediately and processed as shown in Figure 8. In the current implementation, all tasks are treated equally and processed in order. However, in some scenarios, different HTTP requests have different priorities. To achieve the priority feature, a hardware Intel DLB priority queue is used in the NGINX thread pool as shown in Figure 8. The consumer thread fetches tasks from the queue by priority. The priority parameter is configured in the HTTP request Uniform Resource Indicator (URI). The NGINX cache server parses the parameter and sets the corresponding priority. As the latency graph shows, even core usage and efficient hardware scheduling provide latency improvements up to 1.3–2.3X when offloaded to Intel DLB for various object sizes.

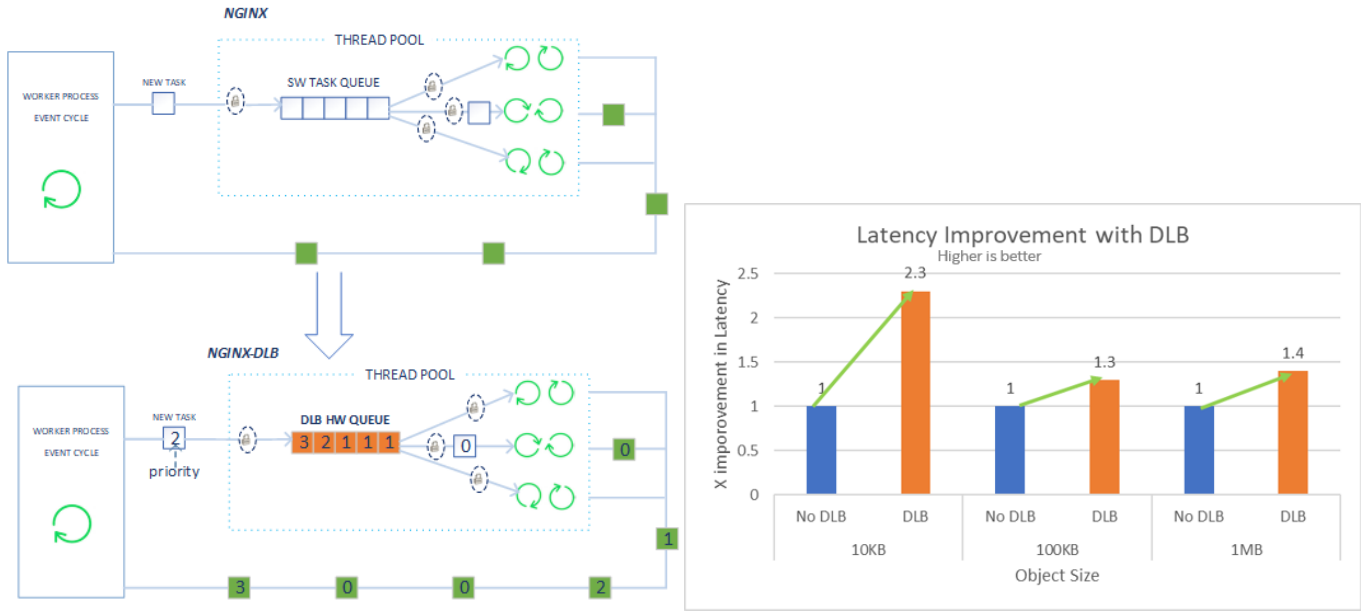

Figure 8. Intel DLB Hardware Queue in NGINX Thread Pool

gRPC

gRPC is a modern, cross-platform, open-source, high-performance remote procedure call technology that enables client and server applications to communicate transparently. gRPC uses protocol buffers to define the services and the messages exchanged between the clients and servers. Figure 9 shows the average performance per core improvement gen-on-gen for a single NUMA zone deployment (1-4 CPU cores) for two gRPC services implemented in Go and C++ and when using small and large messages. Depending on the runtime and on the type of message, the throughput (Requests/Second) improves up to 1.14x for small messages and up to 1.16x for large messages. The latency improves up to 1.14x for small messages and up to 1.16x for large messages.

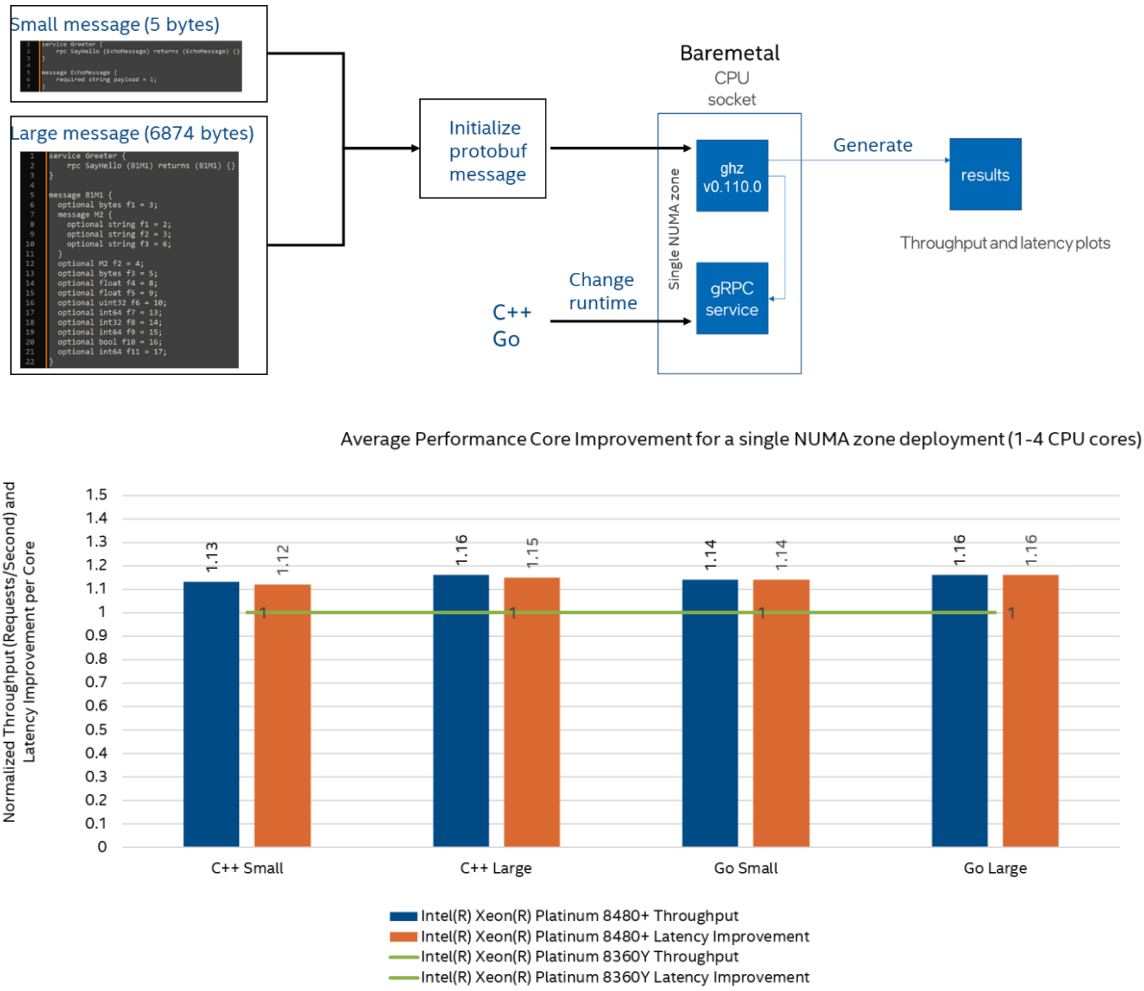

Figure 9. Average Performance per CPU Core Improvement in gRPC for Small and Large Messages

Service Mesh

Service mesh used in microservices environment helps in observability, security, and scalability. Envoy is a popular L7 proxy used in an Istio service mesh. When Envoy is used as an edge proxy, it often must terminate a large number of TLS connections. The RSA asymmetric cryptography operations needed for this can be accelerated using Intel® QuickAssist Technology (Intel® QAT). Intel® QAT is a special hardware accelerator that is visible to the operating system as a PCI device. The Envoy Intel QAT private key provider expects that the Intel QAT devices are available using the regular Linux kernel driver, present in Linux kernel from version 5.15 onward. The Intel QAT endpoint is exposed to Envoy via an SR-IOV VF device, which is the standard Intel® QAT container deployment method used, for example, in Kubernetes via the Intel QAT device plugin.

CryptoMB private key provider uses Intel AVX-512 multi-buffer instructions for accelerating TLS handshakes. The Intel AVX-512 instructions are present starting with 3rd Gen Intel Xeon Scalable processors, and they do not require any special hardware enabling.

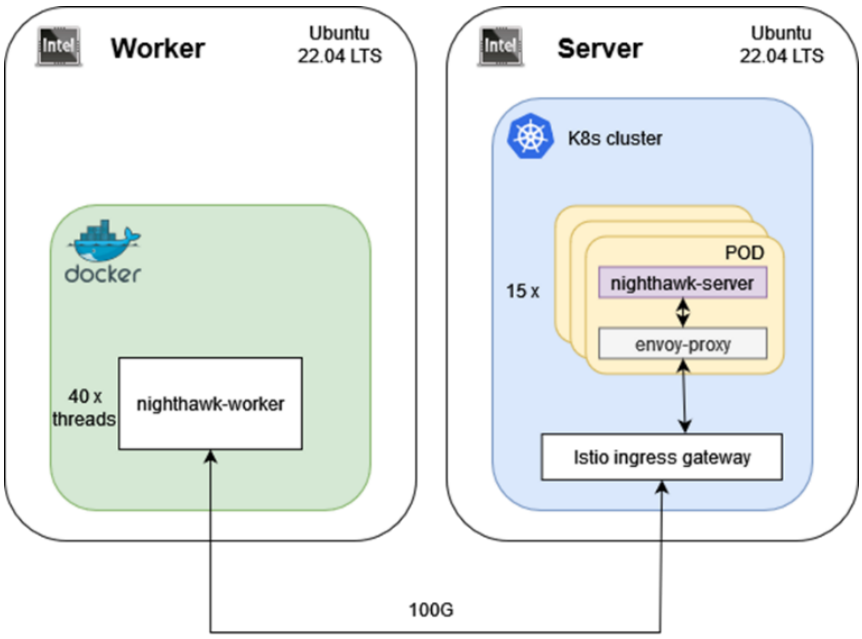

Figure 10. Service Mesh Benchmark Setup

Using a Nighthawk benchmark tool with a client running on one server and an Envoy proxy with microservices as a service mesh as shown in Figure 10, we measure baseline HTTPs with Intel AVX 512 accelerated performance and Intel QAT offloaded performance. CryptoMB private key provider, using Intel AVX-512 multi-buffer RSA acceleration, Intel QAT private key provider, using a single virtual function with (1) TLS v1.3 used with cipher TLS_AES_128_GCM_SHA256, (2) X25519 curve, (3) 2048-bit RSA key. On a 2.0 GHz 4th Gen Intel Xeon Scalable processor we achieved the following results (Figure 11):

- Up to 3.5x throughput improvement with one Intel QAT device and 3x improvement with two Intel QAT devices
- Up to 1.95x latency reduction with two Intel QAT devices
- CPU utilization up to 95% for 1-4 cores and 58% for 8 cores

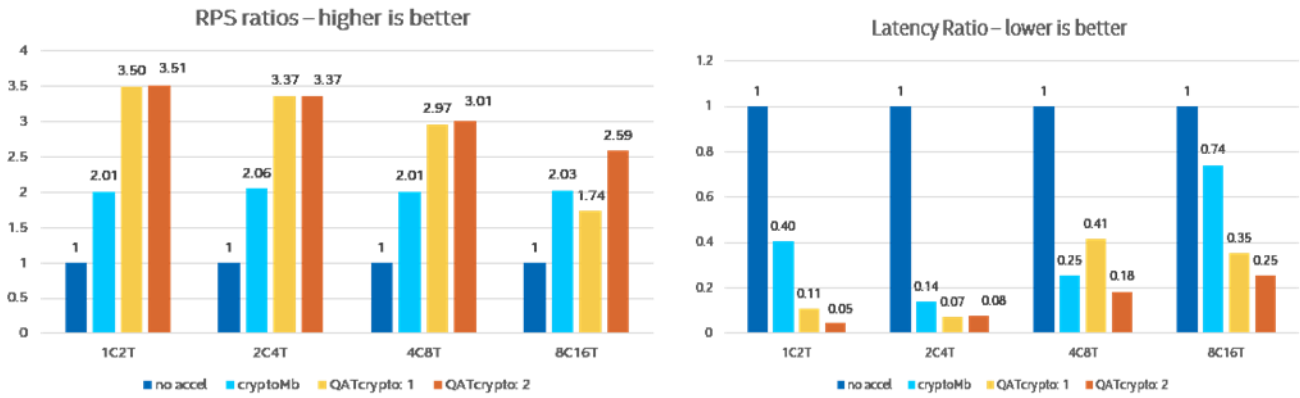

Figure 11. Service Mesh HTTPs Accelerated Performance on 4th Gen Intel Xeon Scalable Processor using Intel QAT

Memcached

Caching for microservices using a Memcached workload can speed up return of results. Intel® successfully optimized performance and scalability on a Pelikan Twemcache workload by leveraging Application Device Queues (ADQ) technology in the 4th Gen Intel Xeon Scalable processor and the Intel® Ethernet 800 Series Network Adapter under the same SLA in a microservices environment achieving a ~3x improvement in throughput and ~10x in P9999 tail latency. Furthermore, a POC web server load balancing on cores in software and using Intel DLB within the 4th Gen Intel Xeon Scalable processor had latency

reductions up to - 22-42% and cycle utilization reduced up to- 30-60%. In addition, the pairing of an Intel® Infrastructure Processing Unit (Intel® IPU) ES2000 with 4th Gen Intel Xeon Scalable processor showcasing the performance differences obtained by moving the microservices load balancer to the Intel IPU ES2000 resulted in a 30% performance improvement between the 3rd Gen Intel Xeon Scalable processor and 4th Gen Intel Xeon Scalable processor. An additional 30% performance increase was observed when the layer 4 load balancer is offloaded to the Intel IPU ES2000.

Conclusion

The world is increasingly complex and the move towards disaggregated architecture remains a challenge. Our results present some of the top challenges for the networking infrastructure layer in microservice performance in the cloud and edge with significant impact to application performance, scaling, and resource utilization. Intel's latest platform, the 4th Gen Intel Xeon Scalable processor with built-in accelerators like Intel Data Streaming Accelerator, Intel Dynamic Load Balancer, and Intel Quick Assist Technology, delivers substantial performance improvements for microservices:

- Up to **2.33x** [MTU 1500, 1500 B] and **2.63x** [MTU 9000, 2400 B] higher single core throughput on 4th Gen Intel Xeon Scalable processor with Intel DSA memory copy compared with software memory copy
- Save **up to 3 cores** to achieve 100 Gbps throughput with 4th Gen Intel Xeon Scalable processor and Intel DSA with DSA memif vs. software memif at MTU 1500 and 1024 B frame size
- Save **up to 2 cores** to achieve 100 Gbps throughput with 4th Gen Intel Xeon Scalable processor and Intel DSA with DSA memif vs. software memif at MTU 9000 and 2400 B frame size
- Up to **3.5x** throughput improvement with one Intel QAT device and 3x improvement with two Intel QAT devices, up to **1.95x** latency reduction with two Intel QAT devices
- Up to **1.14x** improvement for small messages and up to **1.16x** for large messages in a NUMA zone deployment (1-4 CPU cores) for two gRPC services implemented in Go and C++
- Up to **2.3x** latency improvement for CDN workload objects of different sizes using Intel DLB

The outcomes shared describe how the 4th Gen Intel Xeon Scalable processor with built-in accelerators is optimized for microservices applications and demonstrate how Intel is developing innovative solutions for these industry challenges to help our customers efficiently deploy their workloads at scale.

Appendix A Configurations

Envoy: 1-node, pre-production platform with 2x Intel® Xeon® Platinum 8480+ with Intel QAT on Intel ArcherCity with GB (16 slots/ 32GB/ DDR5 4800) total memory, ucode 0x2b0000a1, HT on, Turbo off, Ubuntu 22.04.1 LTS, 5.17.0-051700-generic, 1x 54.9G INTEL SSDPEK1A058GA, 1x Ethernet Controller I225-LM, 4x Ethernet Controller E810-C for QSFP, 2x Ethernet Controller XXV710 for 25GbE SFP28, Nighthawk, gcc version 11.2.0, Docker 20.10.17, Kubernetes v1.22.3, Calico 3.21.4, Istio 1.13.4. DLB SW v 7.8, qatlib is 22.07.1, Nighthawk PODs with response size: 25 PODs each with 1kB/10kB/1MB/mixed size, test by Intel on 10/27/2022.

Calico: 1-node, pre-production platform with 2x Intel® Xeon® Platinum 8480+ on Intel M50FCP2SBSTD with GB (16 slots/ 32GB/ DDR5 4800) total memory, ucode 0x9000051, HT on, Turbo on, Ubuntu 22.04 LTS, 5.15.0-48-generic, 1x 894.3G Micron_5300_MTFD, 3x Ethernet Controller E810-C for QSFP, 2x Ethernet interface, Calico VPP Version 3.23.0, VPP Version 22.02, gcc 8.5.0, DPDK Version 21.11.0, Docker Version 20.10.18, Kubernetes Version 1.23.12, ISIA Traffic Generator 9.20.2112.6, NIC firmware 3.20 0x8000d83e 1.3146.0, ice 5.18.19-051819-generic, Calico VPP Core Number: 1/2/3/4/5/6, VPP L3FWD Core Number: 1/2/3/4/5/6, Protocol: TCP, DSA: 1 instance, 4 engines, 4 work queues, test by Intel on 10/26/2022

gRPC Config1 (ICX) - 1-node, 2x Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz, 36 cores, HT Off, Turbo Off, Total Memory 512GB (32x16GB DDR4 3200 MT/s [3200 MT/s]), BIOS SE5C620.86B.01.01.0004.2110190142, microcode 0xd000375, 1x Ethernet Controller E810-C for QSFP, 2x BCM57416 NetXtreme-E Dual-Media 10G RDMA Ethernet Controller, 2x 1.8T INTEL SSDPE2KX020T8, Ubuntu 22.04.1 LTS, 5.15.0-52-generic, g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, go version go1.17.5 linux/amd64, Ghz v0.110.0, gRPC v1.49, 4838.819 and 4868.116 throughput/core for C++ and Go for Echo message, and 3773.04 and 3785.163 throughput/core for C++ and Go for B1M1 message

gRPC Config2 (SPR) - 1-node, 2x Intel(R) Xeon(R) Platinum 8480+, 56 cores, HT Off, Turbo Off, Total Memory 1024GB (32x32GB DDR5 4800 MT/s [4400 MT/s]), BIOS EGSDCRB1.SYS.8901.P01.2209200243, microcode 0x2b0000a1, 1x Ethernet Controller I225-LM, 4x Ethernet Controller E810-C for QSFP, 2x Ethernet Controller XXV710 for 25GbE SFP28, 1x 54.9G INTEL SSDPEK1A058GA, 2x 372.6G INTEL SSDPF21Q400GB, Ubuntu 22.04.1 LTS, 5.15.0-52-generic, g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, go version go1.17.5 linux/amd64, Ghz v0.110.0, gRPC v1.49, 5496.879167 and 5557.79 throughput/core for C++ and Go for Echo message, and 4392.733333 and 4396.638333 throughput/core for C++ and Go for B1M1 message

NGINX BASELINE: 1-node, Intel(R) Xeon(R) Platinum 8490H 1S 60core HT/ON, Turbo ON, Total Memory 256GB (8 slots/ 32GB/ 4800 MT/s [4400 MT/s]), EGSDCRB1.86B.0091.D05.2210161326, 0xab000110, CentOS Stream 8, 5.15.0-spr.bkc.pc.12.7.15.x86_64, gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-15), ldd (GNU libc) 2.28, nginx version: nginx/1.16.1

See backup for workloads and configurations. Results may vary.

NGINX NEW-1: 1-node, Intel(R) Xeon(R) Platinum 8490H 1S 60core HT/ON, Turbo ON, Total Memory 256GB (8 slots/ 32GB/ 4800 MT/s [4400 MT/s]), EGSDCRB1.86B.0091.D05.2210161326, 0xab000110, CentOS Stream 8, 5.15.0-spr.bkc.pc.12.7.15.x86_64, gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-15), ldd (GNU libc) 2.28, nginx version: nginx/1.16.1, Driver dlb7.8.0v.

Terminology

Table 2. Terminology

Abbreviation	Description
8C	Eight Core
ADQ	Application Device Queues (ADQ)
BGP	Border Gateway Protocol
BNG	Broadband Network Gateway
CDN	Content Delivery Network
CNI	Container Network Interface
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
Intel® AVX-512)	Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
Intel® DLB	Intel® Dynamic Load Balancer (Intel® DLB)
Intel® DSA	Intel® Data Streaming Accelerator (Intel® DSA)
Intel® IPU	Intel® Infrastructure Processing Unit (Intel® IPU)
Intel® QAT	Intel® QuickAssist Technology (Intel® QAT)
IP	Internet Protocol
IPAM	IP Address Management
ISA	Instruction Set Architecture
K8s	Kubernetes
MTU	Maximum Transmission Unit
NF	Network Function
RPC	Remote Procedure Call
RPS	Requests per Second
SR-IOV	Single Root Input Output Virtualization
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UPF	User Plane Function
URI	Uniform Resource Indicator
VF	Virtual Function
VM	Virtual Machine
VPP	Vector Packet Processing

References

Table 3. References

Reference	Source
Application Device Queues (ADQ) Resource Center	https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/adq-resource-center.html
Intel® Dynamic Load Balancer (Intel® DLB)	https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html

Reference	Source
Envoy	https://www.envoyproxy.io/
NGINX	https://www.nginx.com/
Calico-VPP – Get started with VPP networking	https://projectcalico.docs.tigera.io/getting-started/kubernetes/vpp/getting-started
Introducing the Intel® Data Streaming Accelerator (Intel® DSA)	https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator

Document Revision History

Revision	Date	Description
001	February 2023	Initial release.
002	February 2023	Reorganized first two pages; corrected typo on last page.

0223/DN/WIT/PDF 766820 -002US