# Split gRPC: An Isolation Architecture for RPC Software Stacks


Esteban Ramos
U.C. Santa Cruz
Santa Cruz, CA, USA

Robert Soulé
Yale University
New Haven, CT, USA

Peter Alvaro
U.C. Santa Cruz
Santa Cruz, CA, USA

Pietro Bressana
Intel Corporation
Santa Clara, CA, USA

Edmund Chen
Unaffiliated
USA

Uri Cummings
Unaffiliated
USA

Rui Li
Intel Corporation
Santa Clara, CA, USA

James Tsai
Intel Labs
Santa Clara, CA, USA

Rajit Manohar
Yale University
New Haven, CT, USA



## ABSTRACT

Remote procedure calls are a major contributor to performance variance in distributed systems due to lack of isolation and contention on shared resources. We propose a novel split architecture for a popular RPC framework, gRPC. The design partitions RPC applications into two communicating components: one dedicated to user-implemented business logic, and one dedicated to RPC infrastructure processing. The infrastructure process can be run on a dedicated core or on a smart NIC (e.g., IPU or DPU), providing effective physical isolation and predictable performance. An initial evaluation shows that the split architecture adds modest overhead for average case latency, but allows for lower latency and and higher throughput under host CPU load.



## CCS CONCEPTS

• **Hardware** → Hardware accelerators; • **Computer systems organization** → *Client-server architectures*; • **Software and its engineering** → *Message passing*.

## KEYWORDS

Remote Procedure Call, gRPC, SmartNIC

**ACM Reference Format:**
Esteban Ramos, Robert Soulé, Peter Alvaro, Pietro Bressana, Edmund Chen, Uri Cummings, Rui Li, James Tsai, and Rajit Manohar. 2024. Split gRPC: An Isolation Architecture for RPC Software Stacks. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680484




*APSys '24, September 4–5, 2024, Kyoto, Japan*



ACM ISBN 979-8-4007-1105-3/24/09
https://doi.org/10.1145/3678015.3680484


## 1 INTRODUCTION

Remote procedure calls (RPCs) are major contributors to unpredictable performance in distributed systems. RPC libraries are complex software stacks that combine user-supplied "business logic" with a rich set of "infrastructure" logic that interacts with many different hardware and software components. Inadequate isolation can lead resource contention, resulting in performance volatility and high tail latency [17]. Because performance variance can result in poor user experience and reduced revenue [8, 21, 26], reducing tail latency is an important problem.

A tried-and-true approach to addressing performance problems is to "throw more hardware at it" [7]. After all, although transport offload is a somewhat controversial [22], the recent proliferation of smart NICs, i.e., network interface cards with additional processing units, such as Intel's Mount Evans IPU [20] and NVIDIA's BlueField DPU [4], has led to revived interest in the idea [2, 10, 16, 19].

However, production-quality RPC frameworks, like Google's gRPC [11] and Apache Thrift [1], implement a wealth of functionality beyond traditional transport. A non-exhaustive list of functionality includes message serialization, language interoperability, support for meta-data (e.g, authentication information or telemetry data), and security mechanisms (e.g., TLS for end-to-end encryption). They allow for diverse connection or communication options (e.g., multiplexing streams of a single connection, synchronous vs. asynchronous, etc.) and are extensible, allowing developers to replace major components. Moreover, increasingly, RPC libraries have begun to implement functionality typically associated with a service mesh, such as load balancing.

If we are to leverage smart NICs to accelerate or offload parts of the RPC stack and improve performance predictability, then the key design choice is to determine what logic belongs on the host and what logic belongs on the NIC. A natural starting point would be to identify performance bottlenecks and try to offload that functionality. The already discussed transport offload is an attractive target to avoid the fixed overheads of kernel crossings and redundant data copies. However, transport offload will not help in application regimes in which these fixed overheads are dwarfed by data- and topology-dependent costs in serialization, discovery, and load balancing. The conventional wisdom that serialization costs dominate RPC might lead us to explore point solutions that accelerate serialization [24, 29]. However, these efforts might provide only marginal benefit for applications that use large messages with relatively simple serialization logic, and perhaps no benefit at all for applications that favor small messages.

Therefore, we take a different tack—we "cut" RPC applications around isolation boundaries, with the goal of minimizing interference and tail latency. In Section 2, we start from first principles, and identify potential sources of interference. From this, we derive a fully isolated architecture that completely separates predictable infrastructure logic from arbitrary and mercurial business logic. The resulting design partitions RPC applications into two communicating components: one dedicated to user-implemented business logic, and one dedicated to RPC infrastructure processing. While a few prior systems have explored RPC offload to NICs [15, 25, 30], our design is unique in that it offloads *all* of the existing gRPC C++ library implementation, meaning that the wire-format, security, load-balancing, etc. features are all interoperable with existing gRPC-based applications.

From the application developer's point of view, the change is seamless. We have developed a gRPC compiler that automatically generates the partitioned application from an interface description language (.proto) specification of the RPC service. Our C++ prototype simply requires that application developers include a new header file and link against a new software library. At the same time, our design completely separates all application-independent infrastructure logic into a separate, independent process that can be run in software on another core or, ultimately, on a smart NIC. We expect that, having made this partition, further performance improvements can be made by accelerating some of the infrastructure logic in hardware, either via specially designed hardware or on programmable accelerator pipelines.

We present some initial experiments using HyperProtoBench [12], a set of Protobuf messages that are representative of Google's data center workloads. Overall, our evaluation shows the extra communication overhead due to the split architecture is modest, and in exchange for that overhead, the split design allows the gRPC software to maintain comparable throughput and latency for increasingly large business workloads, demonstrating improved isolation.

| Interference | Fundamental? |
|---|---|
| Business Logic × RPC | No |
| Interrupts × Business Logic | No |
| Interrupts × Host OS | No |
| Host OS × RPC | No |
| Interrupts × RPC | Yes |
| Business Logic × Business Logic | Yes |
| Business Logic × Host OS | Yes |

**Table 1: Sources of Interference**

## 2 SOURCES OF INTERFERENCE

The key to reducing performance variability is reducing contention for shared resources. One obvious strategy for avoiding contention is to place every task on its own isolated physical resource, but this is rarely achievable in practice. In general—and in particular with complex systems such as RPCs—the number of concurrent tasks far outnumbers the number of independent compute resources, requiring multiplexing. Many tasks interact in non-trivial ways (e.g., data and control dependencies), and the costs of these interactions increase when we isolate the tasks. Hence even if we had infinite resources, some components should not be physically separated.

In this section, we enumerate the various sources of interference among components that can lead to performance variability in RPC. For each pair of tasks that can contend for a resource, we consider whether this contention is *fundamental* (i.e., due to a tight coupling between the tasks) or merely circumstantial. From this simple taxonomy we derive the SplitRPC architecture.

**Sources of Interference.** To support RPC-based applications, a variety of tasks must concurrently utilize hardware resources. These include the OS kernel and OS background processes, as well as the threads associated with the RPC stack for transport, connection management, encryption, and (de)serialization. Last, but by no means least, is the application business logic, some of which (i.e., server-side function execution) is scheduled by the RPC infrastructure. Any pair of these tasks can mutually interfere when sharing resources, for example, via preemption on the core or head-of-line blocking on a shared queue. Additionally, in the network-intensive applications characteristic of RPC, interrupts triggered by packet arrival can in principle preempt any running task.

The business logic is a necessary evil; indeed, the entire purpose of the system, all of which is otherwise overhead, is to support it. Unfortunately, the business logic is also completely outside the control of the infrastructure. Its resource

requirements are unknown, and so it is not reasonable to pin it to static or wimpy resources; it cannot be expected to yield cooperatively and so it must be preemptively scheduled.

By contrast, the RPC stack is completely within control of the infrastructure, and could reasonably be specialized for an embedded target with modest compute, such as a smart NIC. RPC tasks can often be pinned to individual core or cooperatively scheduled, simplifying the requirements of the embedded OS and further reducing scheduling variability.

On a typical RPC server, as shown in Table 1, all pairs of the sources of interference discussed above may contribute to tail latency. Adding a pool of dedicated compute resources (e.g., on a smartNIC) permits us to use isolation to avoid contention. Our brief analysis shows that there is a natural partitioning of the RPC stack that fits these non-uniform resources: placing the user-supplied business logic on the host cores while placing the infrastructure code that is responsible for all other aspects of the RPC framework on the lower-powered NIC cores. This architecture splits the traditional monolithic RPC stack in a way that cuts "at the joints" [23] coupling components that need to frequently interact while decoupling components that need not interfere.

**A Cut at the Joints.** We now consider in turn each of the conflicts in Table 1 under such a partitioning. First, our partitioning has separated conflicts that are not fundamental. Because the business logic (BL) and RPC infrastructure (RPC) are physically separated, there is no longer any need for (e.g.) a BL thread to preempt a RPC thread (**BL** × **RPC**). Nor is there any need for a network interrupt to preempt a business logic task (**Interrupt** × **BL**), *or* the OS kernel or background tasks supporting it (**Interrupt** × **OS**).

The remaining sources of interference, however, seem fundamental. The RPC infrastructure, likely the only tasks utilizing the network, should be "close" to the interrupts that concern only them (**RPC** × **Interrupts**). Business logic will conflict with other concurrently-running business logic, for which we can rely on the OS scheduler and abundant preemptible resources (**BL** × **BL**), as well as with the OS functionalities that support it (**BL** × **OS**).

The elephant in the room is the seemingly unnatural separation of the RPC infrastructure from the business logic with which is it tightly coupled in terms of both control (the infrastructure decides *when* BL functions are run) and data (function arguments and returns must be copied). Next, we describes how we optimize this pipeline to realize the benefits of the SplitRPC architecture.

## 3 SYSTEM DESIGN

Our design splits the RPC stack into a business logic process and RPC infrastructure process. This architecture is

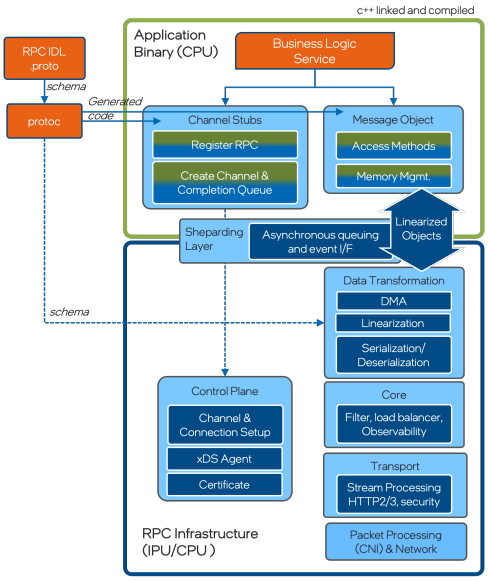



**Figure 1: An overview of the Split architecture.**

amenable to several possible deployments (e.g., to a dedicated CPU core or to various types of smart NICs), but we focus our discssion on an implementation which leverages an IPU. An IPU is an appealing target becuase it is "close to the network", has sufficient computational resources, and is physically isolated from the CPU. Apart from initial setup, only the business logic executes on the CPU. No changes are required to application code.

To offload all the RPC infrastructure processing to a separate IPU process, we must run the gRPC stack on the IPU. The RPC system calls are replaced with stubs that simply communicate the call to the IPU process. When the server starts, we launch a CPU thread that waits for information from the IPU process, executes the business logic, and then sends the reply back to the IPU process.

To support this behavior, we require a separate IPU process that receives the service information/stub calls from the CPU and executes them locally on the IPU. When the service on the IPU starts, we register a *fixed* business logic implementation that communicates inbound RPC messages to the CPU process, waits for a reply from the CPU, and transmits this reply back to the original requester. Finally, we need a communication layer for exchanging data between the IPU and CPU processes.

In summary, the key aspects of the split architecture, illustrated in Figure 1, are: (i) a shepherding layer, used to communicate between the IPU and CPU processes; (ii) a strategy for efficient data transfer between the IPU and CPU; and (iii) a modified RPC compiler plugin that emits the code for the IPU executable that transfers messages to/from the CPU using the shepherding layer.

**Shepherding Layer.** The shepherding layer ensures efficient and reliable communication between the split components of our system. We optimized the design of the shepherding layer to minimize data copies and to allow for concurrent access. It includes two independent communication channels. Each channel is implemented in a separate shared memory segment. The shared memory implements two FIFO ring buffers of a configurable size and four counting atomic head-/tail pointers, two for each ring buffer. The counting atomic head and tail pointers provide both control access to the ring buffers and concurrent access to different slots in the ring buffers. A Boolean flag is coupled to each slot of the ring buffers to mark the slot data valid to read or write.

**Linearization.** In our split architecture, RPC request messages and reply messages have to be transferred between the two different processes: one that handles the RPC infrastructure and the other that handles the business logic. The two processes may have entirely different address spaces; in the case when one process is on the IPU and the other on the host CPU, it is also possible that the IPU and CPU use different underlying instruction set architectures. Hence, a conventional implementation of data transfer between the two halves of the split architecture would copy data from one process to the other. To avoid this extra copy, we introduce an *object linearization* mechanism. The key idea is to transfer the object as a contiguous block of memory that can accessed by the receiver as a valid C++ object, meaning the pointers to the vtable, etc. are all set appropriately. Thus, rather than copying data, there is only a minor "pointer fixup" pass over the data structure, which reduces the cost of data transfer between the IPU and CPU.

**Compiler for Split Architecture.** There are three parts of the overall split RPC system that require code generation using information about the RPC specified in the `.proto` file: (i) object linearization methods for each of the protobuf objects used by a particular RPC; (ii) insertion of object linearization calls at the appropriate points in the RPC stack; and (iii) the creation of the RPC server on the IPU that handles all the RPC processing, and exchanges linearized objects with the CPU process that contains the user-specified business logic. We re-use the existing plugin infrastructure provided by the protobuf compiler to implement the three generators described above.

# 4 IMPLEMENTATION

We have implemented two versions of our split RPC architecture, a functional prototype and a performance prototype.

The functional prototype targets a pre-production silicon of Intel's Mount Evans IPU [5]. Because we used pre-production silicon, we did not have access to drivers for DMA communication. Therefore, we implemented a version of the shepherding layer over TCP using a socket interface, which incurs the obvious overhead of passing through the networking stack twice for every message transfer. We refrain from sharing the performance numbers as they are misleading, given that they simply represent a functional test.

The performance prototype approximates the expected performance on the IPU when the DMA communication path is enabled. The infrastructure and business logic processes are compiled and run on the host server cores, and the shepherding layer is implemented using the System V shared memory API to communicate over a shared memory region. The evaluation in Section 5 use this prototype.

# 5 EVALUATION

Our evaluation shows that (i) the overhead due to inter-process communication in the split design is small (5.2) and (ii) the split design maintains throughput and latency under larger amounts of business logic (5.2).

## 5.1 Methodology

We measure three key metrics—core utilization, throughput, and latency—for an RPC client sending requests to an RPC server, as we increase the amount of business logic performed at the server. We compare two configurations: a baseline with an unmodified RPC server and the performance prototype Split RPC server, in which the IPU logic runs on the host server core. Using `numactl`, the business logic and IPU processes reside on the same socket, but each have dedicated cores connected via the shared memory shepherding layer.

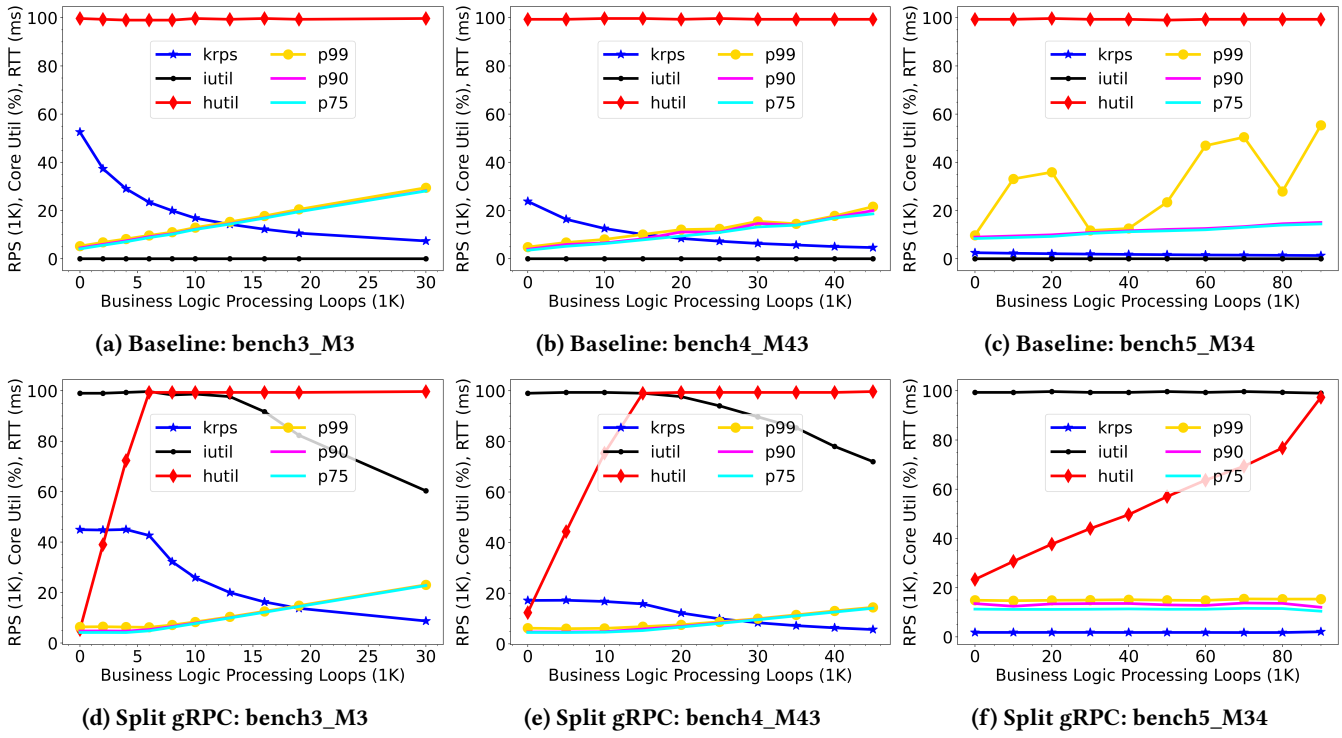
**Testbed.** The client and server were both two socket 32-core Intel Xeon Platinum 8360Y 2.4GHz CPUs with 64GB DDR4 connected via 100Gbps Intel E810-C NICs.

**RPC Workload.** We used Google's representative set of Protobuf messages, HyperProtoBench [12]. To simplify the presentation, we share the results from three messages, that are "small" (1KB), "medium" (48KB), and "large" (476KB): Benchmark 3, Message 3 (bench3_M3); Benchmark 4, Message 43 (bench4_M43); and Benchmark 5, Message 34 (bench5_M34);

**Server Business Logic Workload.** Each service takes an $m$ message as input and returns a different $m$ message as response. To simulate supplementary processing time within the business logic, we introduce an additional configurable processing "busy" loop.

## 5.2 Key Results

The results of our experiments appear in Figure 2. We have presented all of the metrics together to (i) illuminate the overall system trends and (ii) fit the results within the page limits. The primary y-axis is the measurement for the gRPC requests per second in units of 1,000 (KRPS) and the core utilization for the business and infrastructure processing.




(a) Baseline: bench3_M3 (b) Baseline: bench4_M43 (c) Baseline: bench5_M34

(d) Split gRPC: bench3_M3 (e) Split gRPC: bench4_M43 (f) Split gRPC: bench5_M34

**Figure 2: Throughput, latency, and core utilization for gRPC, Split RPC, and Split RPC with a protobuf accelerator.**

The secondary y-axis, on the right, measures the P75, P90 and P99 RTT in milliseconds. The x-axis shows increasing amounts of supplemental business logic processing.

**Cost of Split Architecture** The overhead incurred by the Split gRPC architecture is primarily associated with the shepherding and linearization functions. We measure the overhead using a profiler when the business logic load set to zero, i.e., the host does no work beyond constructing the RPC message instance. For the small message (bench3_M3), less than 25% of host cycles are associated with shepherding. For the large message (bench5_M34) 45% of cycles are attributed to linearization. *In all of these measurements, because the host is doing no real business logic, the overall runtime is very small, so 25% or 45% of host cycles is negligible.*

**Benefits of Split Architecture** Figures 2a, 2b, and 2c display the results for the Baseline configuration on our three representative messages, and Figures 2d, 2e, and 2f display the results of the Split gRPC configuration. In both sets of figures, the red line logs the host core utilization (hutil). For the Baseline, the host core utilization includes the entire business logic and gRPC stack. For the Split gRPC runs, the hutil only includes the business logic. The IPU core utilization (iutil), shown as the black line, logs the utilization of the IPU process running the gRPC infrastructure logic.

We see similar behavior in all three messages. The Split RPC incurs overhead due to linearization and communication, resulting in lower throughput and higher latency when there is no business logic (i.e., the graph's left most points).

However, as we increase business logic processing, we see that the throughput and latency worsen for the Baseline configuration. Because the hutil is saturated, the additional business logic hurts performance. *In contrast, the Split gRPC configurations are able to maintain their throughput and latency under larger amounts of business logic. The split design is able to effectively use the host and IPU processes.* In the figure, we see this as a "knee event", which happens when the business logic is increased to the point that the host core becomes saturated. For larger messages, the "knee" is further to the right in the graphs.

Finally, the Baseline shows a large degree of variance for the P99 latency. Since the hutil is at 100%, the latency can be significantly impacted by system interference, even on a dedicated machine. *In contrast, the Split configuration allows for greater isolation, allowing for more stable latency.*

## 6 RELATED WORK

**Remote Procedure Calls and Optimizations.** Remote procedure calls have existed since the mid-1970s and there is a wealth of prior work on the topic [3, 6, 9, 13, 18, 27, 28].

**Serialization Accelerators.** Optimus Prime [24], Zerializer [29], and the design by Karandikar et al. [14] proposed specialized hardware for serialization. The designs for data transformation in these systems are similar, differing on whether the data transformation logic is in the DMA path, or implemented as a co-processor.

**RPC Accelerators.** HGum [30], Dagger [15] and Cerebros [25] are hardware accelerators for RPCs. However, in contrast to our approach, HGum and Dagger are both inoperable with existing RPC frameworks because they propose their own wire-format for data encoding. Cerebros connects the Optimus Prime [24] serialization accelerator to a TCP offload engine in a NIC. It does not implement the full functionality provided by existing RPC frameworks (e.g., processing of meta-data, streaming connections over HTTP, security, load-balancing, etc.). We evaluate our prototype on a live system, while Cerebros evaluates in a simulation.

## 7 CONCLUSION

Overall, this paper describes an architecture that splits RPC-based applications into separate processes that can be run on separate hardware. The design leverages SmartNICs to reduce tail latency and maintain throughput under larger amounts of business logic, while maintaining interoperability with existing frameworks and without changes to user-code.

## ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation under Grant No. CNS-2212235.

## REFERENCES

[1] Apache Thrift 2021. Apache Thrift. https://thrift.apache.org.
[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 93–109.
[3] Andrew Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (1984), 39–59.
[4] Bluefield 2023. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.
[5] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. 2021. Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU). In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–16.
[6] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. 2003. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI '03)*. 196–208.
[7] Coding Horror 2008. Hardware is Cheap, Programmers are Expensive. https://blog.codinghorror.com/hardware-is-cheap-programmers-are-expensive/.
[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (feb 2013), 74–80.
[9] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. 2003. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*. 175–188.
[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[11] gRPC 2023. gRPC. https://grpc.io.
[12] HyperProtoBench 2022. HyperProtoBench. https://github.com/google/HyperProtoBench.
[13] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.
[14] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. 462–478.
[15] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). 36–51.
[16] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.
[17] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.
[18] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. 295–304.
[19] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 399–413.
[20] MEV 2022. New Intel Mount Evans IPU ASIC DPU at Intel Vision 2022. https://www.servethehome.com/new-intel-mount-evans-ipu-asic-dpu-at-intel-vision-2022/.
[21] Pulkit Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*.

[22] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03)*. 5.

[23] Plato. 1952. *Plato's Phaedrus.* Cambridge : University Press. [265e].

[24] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. 1203–1216.

[25] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. 407–420.

[26] Radar Theme: Web Ops 2008. Radar Theme: Web Ops. http://radar.oreilly.com/2008/08/radar-theme-web-ops.html.

[27] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 1–15.

[28] Brent B Welch. 1986. *The Sprite Remote Procedure Call System.* Technical Report. University of California at Berkeley.

[29] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. 206–212.

[30] S. Zhang, H. Angepat, and D. Chiou. 2017. HGum: Messaging Framework for Hardware Accelerators. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.