NotNets: Accelerating Microservices by Bypassing the Network

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

Matthew Adiletta
Intel Corporation
Santa Clara, CA, USA

David Cheng
Yale University
New Haven, CT, USA

Adrian Cockroft
OrionX
USA

Frank Hady
Intel Corporation
Santa Clara, CA, USA

Ramesh Illikkal
Intel Corporation
Santa Clara, CA, USA

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

Robert Soulé
Yale University
New Haven, CT, USA

Abstract

Remote procedure calls are the workhorse of distributed systems. However, as software engineering trends, such as micro-services and serverless computing, push applications towards ever finer-grained decompositions, the overhead of RPC-based communication is becoming too great to bear. In this paper, we argue that point solutions that attempt to optimize one aspect of RPC logic are unlikely to mitigate these ballooning communication costs. Rather, we need a dramatic reappraisal of how we provide communication. Towards this end, we propose to emulate message-passing RPCs by sharing message payloads and metadata on CXL 3.0-backed far memory. We provide initial evidence of feasibility and analyze the expected benefits.

CCS Concepts

• **Networks** → **Data center networks**; • **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → *Cooperating communicating processes*.

Keywords

Remote Procedure Call, Shared Memory, Microservices, CXL

ACM Reference Format:
Peter Alvaro, Matthew Adiletta, David Cheng, Adrian Cockroft, Frank Hady, Ramesh Illikkal, Esteban Ramos, and Robert Soulé. 2024. NotNets: Accelerating Microservices by Bypassing the Network. 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/10.1145/3678015.3680494

This work is licensed under a Creative Commons Attribution International 4.0 License.
APSys '24, September 4–5, 2024, Kyoto, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1105-3/24/09
https://doi.org/10.1145/3678015.3680494

1 Introduction:

A majority of businesses [1] now use microservice-based architectures for building large-scale applications. Breaking a system into autonomous services that communicate via a fixed API allows development teams to work independently in every sense, implementing their services in any way they want while interacting across services only at the interface level. Microservices also provide scaling benefits along several dimensions: operators can scale individual services independently to accommodate load and bottlenecks, while managers can scale development and support teams for individual services independently.

These organizational and operational advantages come at a profound performance cost. Even looking beyond the more shocking recent headlines, [2] there is increasing evidence that the fundamental costs of microservices may not justify their flexibility. In a typical microservice-based application, a single request flow may trigger hundreds or even thousands of remote procedure calls (RPCs), which incur data serialization, kernel crossing, packet processing, queuing delays, and myriad other resource costs and sources of latency. Facebook reports [37] that only 40% of the compute cycles contribute to processing business logic, with the rest being spent on communication.

To mitigate these ballooning, communication-related overheads we must focus on RPC, the workhorse of microservices. However, reducing overhead without abandoning any of the

[1] 52%, according to a 2020 survey [32]
[2] e.g., Amazon Prime Video's move to a "monolithic" application that saved them 90% in infrastructure costs [1]

generality that made the architecture attractive is a tradeoff that seems difficult to navigate. Many point solutions in the literature target perceived bottlenecks in the RPC stack, including kernel-bypass networking to reduce data copies and kernel crossings [5, 9, 21, 28, 38] and hardware acceleration of (de) serialization [22, 33, 39]. Unfortunately, as we show in Section 2, modern RPC stacks are highly sensitive to workload characteristics. Our experiments, using both client/server benchmarks and simulation of entire microservice-based applications, reveal that minor variations in communication pattern can easily shift the bottleneck from overheads in data transformation, to kernel network stack, to HTTP header processing, to transport-level security and load balancing. Time will not be well-spent on point solutions that target and accelerate a single perceived bottleneck of the communication stack. The problem is the communication itself.

We advocate something more disruptive. Memory has been undergoing a transformation similar to the one experienced by storage a decade ago. RDMA began to show how memory semantics might transcend the boundaries of a host; now technologies such as Compute Express Link [8] (CXL) allow multiple sockets on multiple nodes to access pooled remote memory via a fast interconnect. The next step—sharing the data in the memory "pool" among those nodes, at low latency and without queuing effects—is within reach. This vision may seem perilously close to distributed shared memory (DSM), an old idea that, each time it comes back around, is dismissed by the systems community as impractical due to the challenges of scaling coherence and tolerating faults. We observe that the semantics of RPC, which involve the round-trip transfer of (immutable) data and control between agents that are already assumed to fail independently, places very modest constraints on the sharing medium, and requires none of the complexity of general DSM.

In this paper, we present `Notnets`, a network-bypass strategy that can be retrofitted to existing RPC frameworks. `Notnets` will allow a collection of hosts (with a radix of as many as 512-1024 cores) to use a pool of CXL-attached memory to transparently implement message-passing semantics in a way that avoids all of the dominant bottlenecks in the current RPC stack. By exploiting message passing semantics, `Notnets` is compatible with a variety of hardware- and software-based coherence mechanisms, allowing us to postpone a choice of mechanism until more is known about their peformance characteristics in CXL-based devices. *Our initial experiments suggest that network bypass can improve RPC latency by an order of magnitude.*

2 The Overhead of Microservices

To better understand what factors contribute to the performance overhead of RPCs, we performed a basic experiment with RPC client-server pairs in which the business logic is to simply "echo" the input message. The client and server machines were directly connected to a single switch and no other traffic was sent over the network. Thus, network propagation time was negligible. We performed this experiment on a diversity of messages. To illustrate our observations, we report results for the "hello world" message from the gRPC tutorial [13] and message 54 from bench5 (**m54**) of Google's Hyperprotobench benchmark[16]. We used Intel VTune to capture the call stacks on the gRPC server side, and break down the delay into four categories: gRPC Serialization which mainly captures the time spent on serializing/deserializing the messages, gRPC Transport which mainly involves HTTP header processing for the messages, gRPC Core which consists of other gRPC internal processing including setting up a bunch of internal data structures and handling IOs, and kernel stack which is the TCP/IP stack used to receive/send messages. The results of this initial experiment are shown in Figure 1a.

These microbenchmarks provide initial evidence that RPC stacks are highly sensitive to workload characteristics such as payload size. Factors (e.g., kernel networking or data serialization) that dominate in some scenarios (e.g., m54 receive and send, respectively) are negligible in others (e.g., helloworld receive and send, respectively). This suggests that point solutions targeting perceived bottlenecks in the stack are unlikely to bear fruit in mixed workloads. For example, kernel-bypass

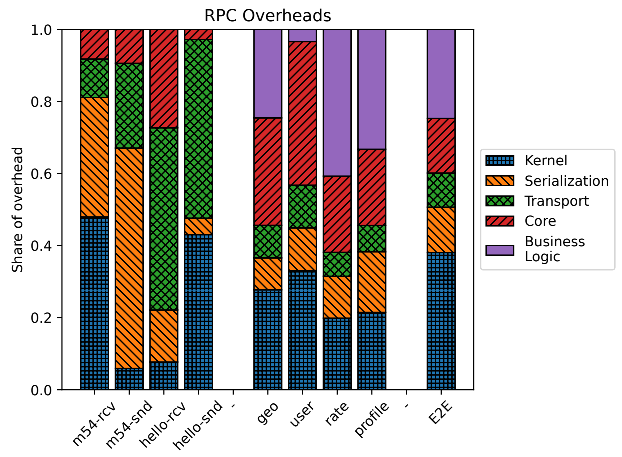

Figure 1: Where is time spent in systems that use RPC? The first four bars (a) draw from benchmarks in Hyperprotobench, while the second (b) profile individual microservices from the `HotelReservation` application in DeathStarBench. The last bar is the combined profile across all services in the `HotelReservation`.

networking is an attractive solution to avoid the fixed overheads of kernel crossings and redundant data copies [5, 9], but at the cost of significant complexity. Efforts to port existing RPC frameworks to utilize the flavor of the month in userspace networking will not be well-spent in application regimes in which these fixed overheads are dwarfed by data- and topology-dependent costs in serialization, discovery, and load balancing. As a second example, the conventional wisdom that serialization costs dominate RPC might lead us to explore solutions that accelerate serialization with specialized hardware [22, 33, 39]. 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.

Of course, these microbenchmarks might not reflect the balance of RPC overhead in practice. What is more, they only study overhead in the RPC stack without considering to what degree this overhead interferes with business logic in practice. To get a clearer picture of RPC overhead in the context of a realistic microservice-based application, we performed a second profiling experiment using the social network application in Deathstarbench [10]. We used the `HotelReservation` application, consisting of 8 services as well as a persistent backing store, using the `mixed-workload_type_1` benchmark. We ran the workload for 30 minutes, and use golang's `pprof` package to profile the results, which are shown in Figure1b.

In addition to plotting time spent in the four RPC categories used in the previous experiment, we include the time spent in user-supplied business logic code. We show the breakdown for four services (`geo, user, rate, profile`) as well as the cumulative breakdown across the entire application. We note first that across all services, only about 25% of CPU cycles are spent doing useful work. We are not too surprised to see an even lower figure compared to that of Facebook's reports, since the application code in the Deathstarbench applications are very simple. As we observed in the microbenchmarks, the overhead of different components of the RPC stack is very sensitive to the workload presented by each individual service. Finally, RPC overheads are balanced across the entire application (as shown in the `merged` bar), with time spent in the kernel dominating slightly.

3 Enabling Trends and Feasibility

The two observation behind this work are: (i) shared, remote memory is possible and almost here, and (ii) adopting RPC semantics allows `Notnets` to sidestep many of the traditional problems associated with DSM.

3.1 CXL and Coherence

Over the past decade, industry has increasingly adopted data center disaggregation [11, 18, 23, 25, 27], which allows storage and compute resources to be scaled independently, improving utilization and efficiency.

Memory is undergoing a similar transition [2, 9, 14, 29, 30]. CXL [8] is a multi-protocol interconnect standard that builds on the PCIe interface to provide remote memory access between a host processor and a device connected over a CXL link. Early use-cases [3, 34, 40] focused on scenarios in which remote memory is dynamically allocated to a particular application. The CXL 3.0 standard, however, brings cache coherence to a shared multi host environment, permitting remote hosts with distinct coherence domains to share access to the same cache line.

CXL Type 3 Devices with support for multiple host/peer interfaces [8] will utilize the hardware based *back-invalidate* coherence mechanism, which ensures that individual hosts obtain exclusive access to cache lines before modifying them In this modality, remote memory will be transparent to applications. An alternative is software-managed coherence, in which applications are responsible for explicitly invalidating and/or flushing cache hierarchies, trading complexity for a reduction in coherence traffic. It is too soon to know which approach will perform best for our system. Our design will support both.

3.2 Is This DSM?

This vision may sound like distributed shared memory (DSM) by another name. DSM, of course, has been studied for decades [4, 6, 7, 26], and it would be natural to question why any new attempt to revive this technology would succeed.

Historically, there have been three main problems that have hindered the adoption of DSM:

- *Access Latency.* Applications that are not written to tolerate non-uniform memory access latencies have unacceptable performance when some memory accesses are remote.
  We discuss how RPC over CXL-backed memory can side-step these issues below.
- *Failures.* Applications are not written to handle partial failures of memory [36], and masking the failure of memory nodes via redundancy [24, 35] incurs unacceptable costs on the critical path of loads and stores.
- *Coherence and Synchronization.* Protocols that manage transparent access to copies of shared data do not scale well [15, 31].

3.3 Why RPC is Different

`Notnets` can side-step all three of these problems because it does not need to support arbitrary, transparent access to shared memory. All that is required is support for the basic semantics of RPC, which narrowly extends the standard single-machine procedure call abstraction to provide transfer of control and data across a computer network. When the remote procedure is invoked, the caller suspends its execution, passes the parameters across the network, and executes the procedure on the callee. When the procedure completes, the results are passed back to the caller. We note these key aspects of the semantics of RPC:

(1) It assumes latency due to message passing.
(2) It assumes that agents fail independently.
(3) It assumes that a client and server own their own copies of data.

Access Latency. There are two key arguments to make about performance. First, distributed applications are already written in such a way that they expect increased latency for remote access due to the overhead of RPC communication. Thus, increased latency is not really an issue when replacing RPC with shared-memory communication. Second, even if it were, the latency is likely better with `Notnets` than with RPC. The performance gap between inter- and intra-node communication has tightened significantly since the early work on DSM. Referencing publicly reported numbers from manufacturers and recent research results [17, 19, 20], we see that although RPC has 100× higher latency than accessing local DRAM, RPC is also 100× faster than accessing local SSD. Recent work [12] reports that DirectCXL memory pooling achieves around 7× better performance than RDMA-based memory pooling.

Failures. Unlike transparent DSM, the RPC model already assumes that agents can fail independently and has well-defined semantics in such contingencies. Unlike a `load` or `store`, a RPC call can return an error to the caller, either due to an explicit error return from the server or (in the event that the server node is down or otherwise unreachable) a client-managed timeout. A typical microservice is designed to anticipate and mitigate the effects of failures of services upon which it depends, either by retrying, taking a fallback path, or supplying a static default. Hence `Notnets` completely sidesteps the problem of transparent fault-tolerance for remote memory.

Coherence and Synchronization. The last bugbear of DSM is the performance and scalability of coherence. As we have observed, the RPC abstraction does not require transparent coherence between concurrently-accessed copies of memory. In RPC, data movement is always explicitly triggered by the application, at which time the coherence state of the data (the immutable "message" being modeled in shared memory) transitions atomically from exclusively owned by the sender to read-only and exclusively owned by the receiver.

4 Notnets: the Potential

Realizing the advantages of our hypothesized shared memory RPC will not be as simple as waiting for vendors to deliver data center shared memory. RPCs provide more than a location abstraction within the data center, and many mechanisms that cannot simply be bypassed will need to be rethought to best take advantage of a shared memory implementation.

In the section below, we discuss the low-hanging fruit for which performance advantages can be shown (as we demonstrate in Section 5.2) by simply *doing less*. Next, we discuss stretch goals that will require further design, and potentially, changes to the programming model to realize.

Kernel Stack. A traditional RPC incurs two kernel crossings (single-digit μs) and at least as many copies of the data payload; the use of out-of-process sidecar proxies (e.g. Envoy), which is common in microservice architectures to handle cross-cutting concerns including discovery and routing, effectively doubles both, and network interfaces may incur additional copies. `Notnets` will avoid all kernel crossings.

L7 (HTTP) Networking. gRPC uses HTTP as a transport mechanism to allow for streaming requests, i.e., so that applications avoid the overhead of opening/closing connections. However, our shared-memory deployment will obviate the need for traditional network communication, and therefore, make the need for L7 networking unnecessary. We do note, though, that gRPC uses HTTP headers to pass meta data (e.g., telemetry information) between the client and server. This mechanism will need to be replaced by a shared-memory implementation, which should be straightforward to implement in a manner similar to passing the RPC payload.

gRPC Data Transform, i.e., (De)serialization. There is no free lunch, and it will not in general be possible to simultaneously avoid all serialization costs and support existing microservice-based applications in their full variety. One of the touted benefits of using microservices is that they permit decoupling (and hence autonomy and independent scaling) of development teams along API boundaries. This autonomy implies that applications can (and often will) be polyglot, in the sense that cooperating services are implemented in separate languages, frameworks, and runtime environments.

While it may be reasonable in practice to make some assumptions about memory representation (e.g. endianness), some serialization cost seems fundamental when sharing values (e.g., floating point numbers) between caller and callee. Endpoints written in different languages and hosted on different platforms could in principle choose a common representation (e.g., Apache Arrow) for data intended to be shared as RPC arguments or returns, obviating the need for (de)serialization. Further study of microservice-based applications are required in order to understand how pervasive polyglot systems are. In any case, fast paths can be explored whenever two adjacent services in the call graph share a common representation, and this can be determined statically.

4.1 Open Questions

Transport Layer Security. On a single server, security (i.e., privacy) is provided via process isolation; the virtual address space in one process is completely separate from the virtual address space in a second process. In other words, process A cannot access memory in process B. The isolation is enforced by the memory management unit (MMU). In particular, a process is not able to directly access physical memory, but rather must use virtual memory addresses that the MMU translates to physical address. In this way, the MMU can ensure that process A cannot "name" addresses in process B.

In a distributed setting, multiple processes, which may reside on physically separate machines, work together to provide the application functionality. Each of these processes have isolated virtual memory spaces. To copy data from one address to another, applications have typically relied on message passing. Transport Layer Security (TLS) encrypts the data to ensure that the data is kept private while in transit.

However, with a shared-memory backend for communication, the security model changes. The communicating processes already share an address space. This suggests that, rather than end-to-end encryption, a new mechanism is needed to ensure isolation of the shared memory. There must be something equivalent to the MMU that ensures that non-participating processes cannot "name" the addresses in the shared memory segment.

Load Balancing and DNS Resolution. Load balancing is needed to spread the workload evenly across the additional machines. DNS is often used as a mechanism to discover available peers. Today, many microservice deployments rely on side-car proxies to interpose on RPC requests and perform load balancing. However, recent versions of gRPC include support for "proxyless" service meshes, in which gRPC can directly process requests from the control plane using the XDS API. As with traditional RPC deployments, our shared-memory approach will adopt a scale-out approach in which we increase capacity by adding additional servers to the memory pool. Thus, we will need to maintain some of the same infrastructure to monitor load and determine which CPU to target for execution. To balance load, we expect to extend gRPC's "serverless proxy" functionality to allow for selecting peers from within the memory pool based on the load information that we collect. We expect that there will be some performance gains compared to standard DNS-based distribution of servers, but the extent of that benefit would need to be experimentally evaluated.

5 Evidence of Feasibility

By cutting the network out of the distributed system, `NotNets` may seem to be proposing to boil the ocean. Nevertheless, this discussion suggests an incremental path. Our initial prototype transparently short-circuits the overheads of kernel and layer 7 networking and TLS. It makes no attempt to side-step serialization overheads, although the next prototype will exploit a fast path when client and server share a common memory representation (a property known when servers are deployed). We now briefly describe our prototype alongside initial evidence of the promise of `NotNets`.

5.1 Network Bypass

Ultimately, `NotNets` will emulate message-passing by sharing message payloads and metadata on CXL 3.0-backed far memory. While we wait for this (or a similar) technology to become available we are path-finding. The initial prototype, designed to answer basic questions about required functionality and best-case performance, runs on a single host using System V shared memory. The communication channel is realized as a circular buffer; we interpose on gRPC's request/response API, using the "custom channel" extension mechanism, to enqueue and dequeue messages at the client and server, respectively.

The choice of a coherence mechanism will have significant performance consequences that are difficult to reason about at this time. The transparent, hardware-based coherence that CXL will provide, in addition to simplifying the design, will avoid costly polling of remote storage, but at the cost of coherence traffic that will otherwise be unnecessary due to the simplicity of the sharing model. To avoid overfitting to a performance assumption that cannot yet be validated, we are maintaining a variety of implementations of the circular queue that make different assumptions regarding coherence and availability of synchronization mechanisms. In this small experiment, we showcase a lock-free queue implementation that uses atomic variables to synchronize.

The prototype bypasses all communication-related overhead including the TCP and HTTP stacks just as `Notnets` will. It also short-circuits away functionality in gRPC related to load balancing, transport-level security, discovery, and other features that our ultimate solution must somehow address, as we discussed in Section 4.

5.2 Evaluation

	avg	p99	p999
http2	209	507	1430
notnets	26.2	138	335
notnets-	9.67	44.1	231

Table 1: Latency in μs.

Our initial feasibility experiment focuses on end-to-end RPC latency, supporting the intuition that you can do things a lot faster if you do a lot less. We use a single Google Cloud Platform `ec-standard-4` host, configured with 4 vCPUs and 16GB memory, to host the server and client, and reproduce the "HelloWorld" experiment reported in Section 2. We measure the latency of a trivial "echo" RPC end-to-end from invocation to completion on the client, in three scenarios. **http2** uses the standard http-based transport of GRPC. **notnets** uses the prototype described in Section 5.1. **notnets-** uses the prototype without serializing the message payload.

Table1 reports latencies in microseconds, showing near an order of magnitude performance gain short-circuiting away the RPC. Bypassing serialization can offer another 3× improvement. We should do this.

6 Conclusion

Systems engineering is characterized by tradeoffs, and it is a rare and happy day when we can have our cake and eat it too. Nevertheless, the emerging systems landscape, driven by other concerns (in this case, saving money by eliminating stranded memory), has offered us a unique opportunity. We can dip our toes into DSM and enjoy *only* its benefits, postponing its downsides for future research.

Acknowledgments

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

References

[1] 2023. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Article 14, 16 pages.

[4] R. Bisiani and M. Ravishankar. 1990. PLUS: a distributed shared-memory system. In *Proceedings. The 17th Annual International Symposium on Computer Architecture*. 115–124.

[5] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K. Panda. 2018. Accelerating TensorFlow with Adaptive RDMA-Based gRPC. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 2–11.

[6] Nicholas Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. 1994. The Linda® Alternative to Message-Passing Systems. *Parallel Comput.* 20, 4 (1994), 633–655.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. 152–164.

[8] Compute Express Link 2023. Compute Express Link. https://www.computeexpresslink.org.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 3–18.

[11] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 249–264.

[12] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling With CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[13] gRPC 2023. gRPC. https://grpc.io.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[15] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149.

[16] HyperProtoBench 2022. HyperProtoBench. https://github.com/google/HyperProtoBench.

[17] Intel NVMe with 3D XPoint Technology chart 2015. Intel NVMe with 3D XPoint Technology chart. https://www.tomshardware.com/reviews/intel-micron-3d-xpoint-updates,4286.html.

[18] Intel Rack Scale Design (Intel RSD) 2018. Intel Rack Scale Design (Intel RSD). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[19] Intel Skylake 2019. Intel Skylake. https://www.7-cpu.com/cpu/Skylake.html.

[20] Intel Xeon Processor E7-8893 v3 2019. Intel Xeon Processor E7-8893 v3. https://ark.intel.com/content/www/us/en/ark/products/84688/intel-xeon-processor-e7-8893-v3-45m-cache-3-20-ghz.html.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. 295–306.

[22] 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.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 29, 15 pages.

[24] Leslie Lamport. 1998. The Part-time Parliament. *ACM TOCS* 16, 2 (May 1998), 133–169.

[25] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. 2017. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[26] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. 1998. The DASH Prototype: Implementation and Performance. In *25 Years of the International Symposia on Computer Architecture (Selected Papers) (ISCA '98)*. 418–429.

[27] Chung-Sheng Li, Hubertus Franke, Colin Parris, and Victor Chang. 2015. Disaggregated Architecture for at Scale Computing. In *ESaaSA 2015 - Proceedings of the 2nd International Workshop on Emerging Software as a Service and Analytics, Lisbon, Portugal, 20-22 May, 2015*, Victor Chang, Muthu Ramachandran, Gary B. Wills, Robert John Walters, Verena Kantere, and Chung-Sheng Li (Eds.). 45–52.

[28] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. 2021. HatRPC: Hint-Accelerated Thrift RPC over RDMA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Article 36, 14 pages.

[29] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. 267–278.

[30] Pankaj Mehra and Tom Coughlin. 2022. Taming Memory With Disaggregation. *Computer* 55, 9 (2022), 94–98.

[31] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. 2020. *A Primer on Memory Consistency and Cache Coherence* (2nd ed.).

[32] O'Reilly 2020. Microservices Adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[33] 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.

[34] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[35] F. B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* 22 (Dec. 1990), 299–319.

[36] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-scale Field Study. *PER* 37, 1 (June 2009), 193–204.

[37] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale *(ASPLOS '20)*. 733–750.

[38] 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.

[39] 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.

[40] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 55–71.