Disaggregated Applications Using Nanoservices

Xinwen Wang
Cornell University
xinwen@cs.cornell.edu

Yu-Ju Huang
Cornell University
yh885@cornell.edu

Tiancheng Yuan
Cornell University
ty373@cornell.edu

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

Abstract

The prevailing approach toward building software infrastructure for disaggregated data centers is to develop new virtual machine monitors or operating system kernels that present the underlying hardware as a collection of *logical servers*. While this works well for backward compatibility, we argue that it might be better to invest in rebuilding the applications themselves so they themselves are disaggregated along physical boundaries. We propose *nanoservices*, a type of microservice that is highly specialized for a single type of hardware resource, and that applications be built from such nanoservices, communicating over fast interconnects. This approach keeps operating systems simple while taking better advantage of available hardware resources.

1 Introduction

A desire for improved utilization of hardware resources is driving a trend toward disaggregated data centers. Instead of having racks filled with *one-size-fits-all* servers, each containing a collection of CPUs, memory, storage, NICs, GPUs, FPGA, and so on, the idea of a disaggregated data center is to fill racks with pools of specialized servers and connect these with a fast interconnect such as RDMA [23], Gen-Z [3], or CXL [1]. So a rack will contain pools of CPU servers, memory servers, NIC servers, disk servers, GPU servers, and more. The software approach is to create *logical servers* specialized for the applications that run on them. However, a recent study [28] shows that hiding underlying disaggregation can cause significant performance degradation for unmodified applications. In this paper, we argue that creating logical servers may not be the right one to best utilize disaggregated resources. Like [9] we argue that hardware resource disaggregation should be exposed to applications rather than hidden from them. However, we go significantly further and propose that the applications themselves should be disaggregated as well.

We propose to build applications as a collection of *nanoservices*. A nanoservice predominantly uses a single type of resource. Today, a cloud application may be structured as a set of microservices, but each microservice typically requires a variety of resource types. For example, NGINX [4] requires CPUs, memory, NICs, and more. Even a microservice like memcached is a networked service that requires powerful CPUs and NICs to run well. We believe those microservices should be further decomposed into nanoservices. For in-memory caching, we need a version of memcached that uses purely RDMA instead of TCP. A microservice like NGINX can be decomposed into a set of nanoservices, one that deals with TCP connections, one that deals with caching data, and one that deals with storage. The HTTP logic of such a service can also be captured in a nanoservice that mostly needs CPU resources. This nanoservice should orchestrate the handling of HTTP requests, but it should not process or cache the data. The data should flow directly from the caching nanoservice to the TCP nanoservice.

We believe that an application built from nanoservices will better scale in various dimensions than applications built from microservices can. Moreover, it will be easier to deploy disaggregated applications in a disaggregated hardware environment, requiring no special operating system support. New resource types can be readily incorporated. Nanoservices, because of their more specialized nature, may also be easily reusable. The TCP nanoservice can be used by any application that requires interaction with remote clients (beyond RDMA). Like memcached, a memory nanoservice can be used by any application that requires caching. We consider Network Function Virtualization a form of nanoservice that we also want to integrate into our architecture. We want to identify a collection of these reusable nanoservices to simplify the development of disaggregated applications.

But we also need to design abstractions to help developers build applications from nanoservices. For ex-

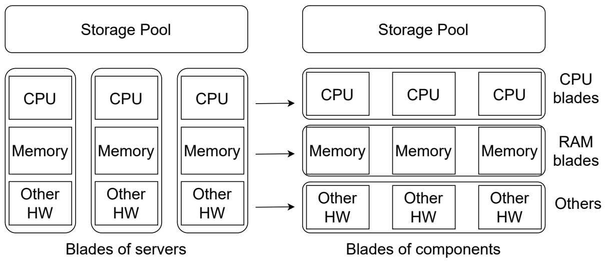

Figure 1: Server-centric v.s. disaggregated architecture.

ample, we will need scatter/gather operations so that we can direct the TCP nanoservice to collect data from various nanoservices and send the result as a single message to a remote client. Due to their distributed nature, we need abstractions that have a clean failure model and make building fault tolerant applications relatively easy. Ideally we make it easy to re-architect existing microservice-based applications into nanoservice-based applications. And finally, we need to make sure that the additional distribution does not come at a significant performance penalty.

Because we do not attempt to hide the underlying disaggregated hardware by presenting it as a collection of logical servers, we do not believe that a nanoservice-based architecture requires the development of new operating systems or virtual machine monitors. Existing operating system kernels such as Linux are already highly configurable. We can customize Linux kernels for disaggregated hardware. If the applications are built from nanoservices, then there is no need for special kernel or VM mechanisms to support running those applications on disaggregated hardware. For performance, we can utilize kernel by-pass techniques such as RDMA, DPDK [2], SR-IOV [5], etc. We do, however, need efficient mechanisms to dynamically redistribute resources among different applications as needed. This may require some changes to the operating system kernel.

2 Background

In a traditional data center, the de facto deployment unit is a server, including processors, memory, and disks on one chassis board. This is the so-called server-centric (aggregated server) design. A disaggregated server design breaks monolithic servers into components of particular types and groups components of the same type together into pools. Figure 1 shows the transformation from a monolithic design to the disaggregated design.

Resource disaggregation provides several benefits over a server-centric design. In a disaggregated environment, different hardware components can be upgraded and maintained independently. Because of the physical boundaries between monolithic servers, certain resources may go underutilized. Disaggregated servers improve resource utilization because resources are globally accessible. As new hardware such as accelerators, GPUs, and specialized hardware, are quickly evolving and becoming available in the data center, the loose coupling between hardware components in a disaggregated environment makes it heterogeneity-friendly.

However, a disaggregated architecture requires a fast interconnect between resources to have comparable performance to server-centric architectures. Emerging device interconnects such as RDMA, Gen-Z, and CXL, can provide bandwidths of 100 Gbps and beyond [11,12,20]. As long as latencies are bounded by a few microseconds, most data center applications can be run on a disaggregated architecture without significant performance loss [15]. Several hardware disaggregation designs have been proposed [10,14,18].

On the software side, LegoOS [24] proposes a split kernel operating system design to manage the underlying disaggregated hardware components while providing a POSIX interface to applications for backward compatibility. To effectively utilize the memory pool, systems such as [6–8,16,19,21] either use new specialized hardware or design new data structures to efficiently leverage remote memory. For disaggregated storage, industry has already moved further in that direction [27]. [25] proposes a way that leverages remote persistent memory such as Non-Volatile Memory (NVM) without requiring a computing unit at the storage server. [9] proposes exposing hardware disaggregation to applications with features like explicit failure notifications and memory grant and steal. [22] points out the similarities between serverless computing and resource disaggregation and proposes that both can be co-designed to shape the future cloud.

In modern cloud applications, application business logic is decomposed into loosely coupled distributed services called microservices [13]. Microservices are designed to be able to scale out easily from one instance to many instances and are commonly deployed using either virtual machines or containers. Unfortunately, this requires complex logical servers when deployed in a disaggregated architecture.

3 Disaggregated Applications

To a large extent, today's cloud applications are already highly modularized. Many new applications are built using paradigms such as serverless computing and microservices. Doing so has various advantages. First, customers who deploy the software need only pay for the resources that they use. Second, the applications become easily scalable, as more resources can be added as needed (elasticity). Finally, because state and function are often separated, it is relatively easy to recover from failures.

However, the applications are modularized and divided into components along *logical* boundaries, not *physical* boundaries based on resources. Thus each component, such as a microservice or a lambda, may still require a full complement of resources. The approach to supporting disaggregated architecture therefore has been to build operating systems or hypervisors that mostly hide disaggregated hardware, presenting the disaggregated hardware as a collection of virtual servers, each appearing as a standard physical server. This is a useful approach, as it provides backward compatibility for existing applications, yet allows each virtual server to be specialized for a particular application process that runs on the virtual server.

While useful, such virtual servers require a lot of mechanisms to hide the underlying distribution of resources. For example, projects like Infiniswap [16] and Leap [19] provide efficient remote paging to hide memory disaggregation. This complicates the operating systems and/or virtual machine monitors while the level of indirection makes them less efficient than if the applications could use underlying physical resources directly.

We can leverage the fact that people are already building modularized applications—we simply need to move the boundaries to coincide with physical reality. We propose building applications from *nanoservices* that are specialized for a particular hardware resource and scale out with only that resource (Figure 2). The nanoservices require a fast communication backbone such as RDMA. Using this approach, we do not need to build operating systems that try to re-aggregate disaggregated resources into logical servers. Instead, we can use a conventional operating system, perhaps optimized for a particular underlying platform. Such platforms might include CPU servers, GPU servers, TPU servers, FPGA servers, RAM servers, NVM servers, SSD servers, NIC servers, and so on. Each will have some customary CPU and memory to be able to run a conventional operating system. For instance, a smart NIC server may have many smart NICs with relatively few CPUs and little memory required to manage them. Applications access the other resources either through system call interfaces or the kernel bypass mechanisms, if available.

Broadly, we can classify nanoservices into two classes: *data-path* and *control-path* nanoservices. Data-path nanoservices focus on data processing and transfer, such as TCP nanoservices and Memcached nanoservices. Control-path nanoservices focus on application logic, such as a load balancing nanoservice or a web server nanoservice. Because nanoservices can scale independently, we expect that applications based on nanoservices can achieve better resources utilization than similar applications running on a logical server box.

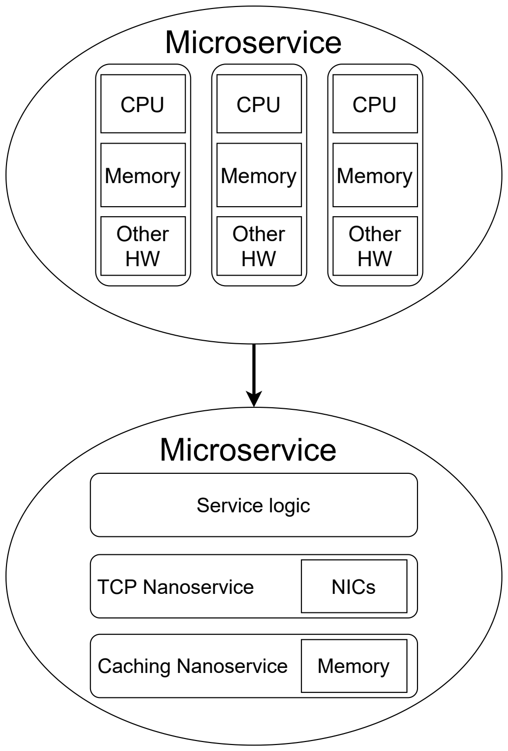

Figure 2: Non-disaggregated and disaggregated application structure.

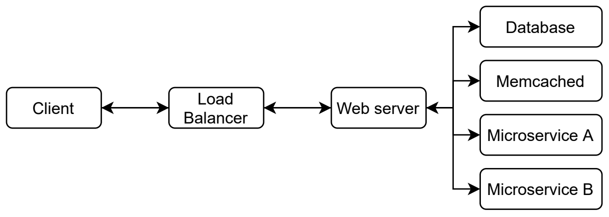

Figure 3: A traditional web server with microservices. Interactions among microservices are not all shown here.

Finally, we note that disaggregated applications are a good fit for serverless applications. Because of their smaller size, a serverless application based on nanoservices can have a lower startup time. Furthermore, a serverless application can be relatively easily transformed into a disaggregated application by matching its functions to nanoservices. Table 1 shows a comparison between microservices and nanoservices.

Case Study 1: A Web Server

In a microservice architecture (Figure 3), a web server is a monolithic application. Such an application requires significant CPU, memory, and networking resources. Just to be able to drive a TCP (or TLS) connection at

	Microservice	Nanoservice
Deployment unit	A logical server, such as a virtual machine or a container	A physical hardware platform with a specialized resource
Modularity level	Coarse-grained, combining various functionalities and using multiple resource types	Fine-grained, focusing on managing a single hardware resource
Network stack	Usually TCP/IP	RDMA or optical switching network

Table 1: Comparison between microservices and nanoservices.

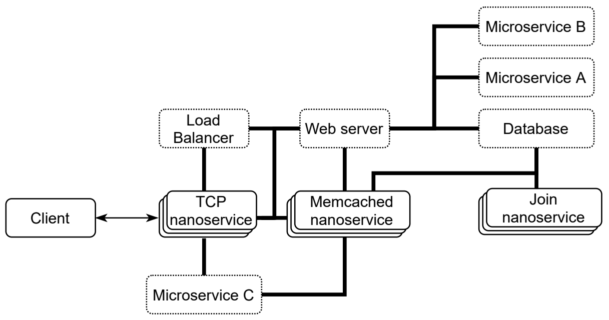

Figure 4: A disaggregated web server microservice with nanoservices. Thick lines are RDMA links and dashed rectangles represent microservice main logic code. Interactions among microservices are not all shown here.

10+Gbps often requires an entire core or more. A web server also needs significant memory for caching. We propose to split a web server up into multiple nanoservices. One or more TCP nanoservices maintain connections to remote clients. Such nanoservices run on server boxes that have many NICs and some CPUs to handle the TCP and/or TLS connections. One or more memory nanoservices are intended to cache data. For dynamically generated web pages, the web server may also deploy nanoservices on CPU or GPU servers. For serving streaming video or audio content, there may be nanoservices that run on storage servers.

One control-path nanoservice running on a CPU server receives the HTTP request headers and orchestrates the subsequent handling of the incoming requests. This nanoservice is the brain of the operation. Ideally, it does not process any user data—using scatter/gather operations, user data is sent directly between the TCP nanoservices and the nanoservices that maintain the data.

By splitting the web server into nanoservices, as shown in Figure 4, we hope to be able to scale the web server in new ways. A monolithic web server cannot handle many client connections, but by separating connection management in a separate nanoservice, and possibly instantiating multiple of these on different NIC servers, we hope to be able to scale with the number of client connections. Similarly, caching resources, streaming resources, and so on can be individually scaled.

By keeping the "brain" nanoservice out of the critical path of data transfer, we hope we can avoid it from becoming a bottleneck. But if it becomes a bottleneck, traditional load balancing techniques allow us to run multiple brain nanoservices. The multiple brains can share their caching nanoservices and other content-serving nanoservices, rather than each being a monolithic web server in its own right without the advantage of a shared cache.

Case Study 2: A Database

For many applications, the database is an essential component that maintains the state of the application. It is often sharded for scale. Each database forms a microservice and requires a multitude of resources, including CPU, memory, and storage, and increasingly also specialized hardware such as FPGAs or GPUs. However, we know how to build shared-nothing distributed database architectures that are highly modularized. We believe those components are already suitable to run as nanoservices. The components include storage nanoservices, query processing nanoservices, and logging nanoservices. Other components may include nanoservices specialized in query optimization, metadata management, map/reduce operations, and so on.

Examples of Reusable Nanoservices

We believe that we can develop a variety of nanoservices that can be highly reusable. Below we will highlight a few examples.

- A cache nanoservice. Such a service would be similar to memcached in functionality. memcached is highly popular in microservice-based architectures. However, memcached deployments live in user space and are accessed over TCP connections. This results in significant access overhead. Instead, we will support memcached-like nanoservices that are directly accessible over RDMA. They would run on servers that have lots of memory but only modest CPU support to run an operating system.

- A persistent key/value store nanoservice. Such a service would provide simple persistent storage. The server that runs the service would have significant storage, but does not need much in the way of CPU or memory resources. We intend to support different kinds of underlying persistent storage. For NVM, we can use pDPM [26] as a starting point.
- When it comes to a full storage stack, designs such as Pocket [17] already go a long way toward what we have in mind.
- A TCP/TLS nanoservice. These nanoservices maintain connections to remote clients, directing traffic as orchestrated by control-path nanoservices. Each application owns a set of remote connections. Then, proxy functionality and load balancing would be achieved by moving a remote connection from one connection set to another. The server running such services could have many smart NICs and some CPUs with a small amount of fast local memory.
- Machine learning training and inference nanoservices. Many machine learning applications are already divided into GPU code and CPU code.

4 Challenges

Developing disaggregated applications and running them efficiently in a disaggregated architecture will come with significant challenges. We list some of these challenges below.

One approach to developing disaggregated applications will be to convert existing microservice-based applications. Currently, most microservices assume to run in a normal server environment with access to a variety of resources. Even the libraries that application processes use make such assumptions. For example, popular programming environments such as Python and Node.js support HTTP processing, but assume that they also have direct access to the TCP connections. While it is possible to create remote sockets over RDMA, this would cause user data to move repeatedly between different nanoservices, an inefficient proposal. We will design new scatter/gather abstractions to avoid such unnecessary copying, but they will likely require changes to existing libraries and other software infrastructure.

Experience demonstrates that it is possible to achieve good performance with microservice-based applications. By further decomposing applications into nanoservices, we add additional communication requirements. While we are shifting such communication to fast interconnects such as RDMA or optical switching fabrics, we must be careful not to introduce communication bottlenecks.

Another potential challenge is security of a nanoservice-based architecture. Because communication has to have ultra-low latency, standard encryption-based security mechanisms may impose too much overhead. Also, nanoservices on the same server may use kernel bypass for good performance, preventing the kernel from mediating access. We plan to use hardware-based isolation mechanisms (for example, SR-IOV [5]) as much as possible to maintain good performance while also providing adequate security. Given that nanoservices will be relatively small and will be running on kernels that can be specialized and minimized to only run the software necessary for the resources available on the server, we also hope to gain security from a smaller TCB.

Another concern is portability, we argue that applications should be disaggregated into nanoservices based on the resources they use, and the resulting nanoservices could become highly specialized, perhaps even depending on specific brands of resources. This could make it hard to move applications from one environment to another. Abstraction remains an important tool to achieve portability. With the right abstractions in place, there could be different specialized implementations of the same nanoservice, providing good performance for the same application running in different environments.

5 Conclusion

To take full advantage of disaggregated data centers, we propose to forego an approach based on logical servers in favor of an approach based on nanoservices, each specialized for a particular type of resource. We believe that applications based on nanoservices can be more easily scaled in various dimensions (based on available resources) and take advantage of new resource types. Today programmers are already writing code that specializes for specific resource types such as GPUs and network processors—we simply propose to extend this to all resource types. We hope to develop a set of reusable nanoservices and communication paradigms to make it relatively easy to develop performant, robust, and secure nanoservice-based applications. We are also looking for approaches to simplify conversion of microservice-based applications to nanoservice-based ones.

For backward compatibility, it may still be useful to also develop logical servers that all but hide hardware disaggregation. But new applications can already take advantage of hardware disaggregation with operating systems available today, configured to take full advantage of the hardware servers on which they are deployed.

References

[1] Compute Express Link. `https://www.computeexpresslink.org/`, 4 2021.

[2] Data Plane Development Kit. `https://www.dpdk.org/`, 3 2021.

[3] Gen-Z Consortium. `https://genzconsortium.org/`, 4 2021.

[4] NGINX. `https://www.nginx.com/`, 3 2021.

[5] Single Root I/O Virtualization and Sharing Specification Revision 1.1. `https://pcisig.com/specifications/iov/`, 3 2021.

[6] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., NOVAKOVIĆ, S., RAMANATHAN, A., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote Regions: A Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 775–787.

[7] AGUILERA, M. K., AMIT, N., CALCIU, I., DEGUILLARD, X., GANDHI, J., SUBRAHMANYAM, P., SURESH, L., TATI, K., VENKATASUBRAMANIAN, R., AND WEI, M. Remote Memory in the Age of Fast Networks. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, Association for Computing Machinery, p. 121–127.

[8] AGUILERA, M. K., KEETON, K., NOVAKOVIC, S., AND SINGHAL, S. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2019), pp. 120–126.

[9] ANGEL, S., NANAVATI, M., AND SEN, S. Disaggregation and the Application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)* (2020).

[10] ASANOVIĆ, K. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, 2014), USENIX Association.

[11] BALLANI, H., COSTA, P., BEHRENDT, R., CLETHEROE, D., HALLER, I., JOZWIK, K., KARINOU, F., LANGE, S., SHI, K., THOMSEN, B., AND WILLIAMS, H. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 782–797.

[12] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow. 9*, 7 (Mar. 2016), 528–539.

[13] CERNY, T., DONAHOO, M. J., AND TRNKA, M. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev. 17*, 4 (Jan. 2018), 29–45.

[14] FARABOSCHI, P., KEETON, K., MARSLAND, T., AND MILOJICIC, D. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[15] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 249–264.

[16] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 649–667.

[17] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[18] LIM, K., CHANG, J., MUDGE, T., RANGANATHAN, P., REINHARDT, S. K., AND WENISCH, T. F. Disaggregated Memory for Expansion and Sharing in Blade Servers. *ACM SIGARCH computer architecture news 37*, 3 (2009), 267–278.

[19] MARUF, H. A., AND CHOWDHURY, M. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 843–857.

[20] NVIDIA. NDR 400G INFINIBAND ARCHITECTURE. `https://www.nvidia.com/en-us/networking/ndr/`, 3 2021.

[21] PEMBERTON, N. Exploring the Disaggregated Memory Interface Design Space. In *The First Workshop on Resource Disaggregation (WORD)* (2019).

[22] PEMBERTON, N., AND SCHLEIER-SMITH, J. The Serverless Data Center: Hardware Disaggregation Meets Serverless Computing. In *The First Workshop on Resource Disaggregation (WORD)* (2019).

[23] RECIO, R., METZLER, B., CULLEY, P., HILLAND, J., AND GARCIA, D. A Remote Direct Memory Access Protocol Specification. Tech. rep., RFC 5040, October, 2007.

[24] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 69–87.

[25] TSAI, S.-Y., SHAN, Y., AND ZHANG, Y. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 33–48.

[26] TSAI, S.-Y., SHAN, Y., AND ZHANG, Y. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 33–48.

[27] VUPPALAPATI, M., MIRON, J., AGARWAL, R., TRUONG, D., MOTIVALA, A., AND CRUANES, T. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 449–462.

[28] ZHANG, Q., CAI, Y., CHEN, X., ANGEL, S., CHEN, A., LIU, V., AND LOO, B. T. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow. 13*, 9 (May 2020), 1568–1581.