Disaggregation and the Application

Sebastian Angel
University of Pennsylvania

Mihir Nanavati
Microsoft Research

Siddhartha Sen
Microsoft Research

Abstract

This paper examines disaggregated data center architectures from the perspective of the applications that would run on these data centers, and challenges the abstractions that have been proposed to date. In particular, we argue that operating systems for disaggregated data centers should not abstract disaggregated hardware resources, such as memory, compute, and storage away from applications, but should instead give them information about, and control over, these resources. To this end, we propose augmenting OSes for disaggregation so as to benefit data transfer in data parallel frameworks and speed up failure recovery in replicated, fault-tolerant applications, as well as discussing some of the implementation challenges.

1 Introduction

Disaggregation splits existing monolithic servers into a number of consolidated single-resource pools that communicate over a fast interconnect [5, 29, 31, 34, 46, 47, 62]. This model decouples individual hardware resources, including tightly bound ones such as processors and memory, and enables the creation of "logical servers" with atypical hardware configurations. Disaggregation has long been the norm for disk-based storage [30] because it allows individual resources to scale, evolve, and be managed independently of one another. In this paper, we target the new trend of memory disaggregation.

Existing works on disaggregated data centers (DDCs) have focused primarily on the *operational benefits* of disaggregation—it allows resources to be packed more densely and improves utilization by eliminating the bin-packing problem. As a result, these works strive to preserve existing abstractions and interfaces and propose runtimes and OSes that make the unique characteristics of DDCs transparent to applications [11, 62]. The implicit underlying assumption in these works is that from the perspective of the OS, the distributed nature of processors and memory is an inconvenient truth of the underlying hardware, much like paging or interrupts, that should be abstracted away from applications.

Our position is that disaggregation is not just a hardware trend to be tolerated and abstracted away to support legacy applications, but rather one that should be *exposed to applications and exploited for their benefit*. We draw inspiration from decades-old distributed shared memory systems (which closely resemble disaggregation) where early attempts at full transparency quickly gave way to weaker consistency and more restrictive programming models for performance reasons [13, 32, 33, 42, 54, 58]. While the rationale for externalizing memory has changed, along with the hardware and target applications, we believe that co-designing applications and disaggregated OSes remains an attractive proposition.

Two properties of disaggregated hardware with potential to benefit applications are the ability to *reassign memory* by dynamically reconfiguring the mapping between processors and memory, and the failure independence of different hardware components (i.e., the fact that processors may fail without the associated memory failing or vice versa). Memory reassignment can be leveraged by applications performing bulk data transfers across the network to achieve zero-copy operations by remapping memory from the source to the destination, or during processor failures to find orphaned memory a new home. Failure independence also allows processors to be useful despite memory failures by acting as fast and reliable failure informers [3] and triggering recovery protocols.

We target data center applications that are logically cohesive, but physically distributed across multiple co-operating instances—examples of these include most microservice-based applications, data parallel frameworks, serverless computing, distributed data stores, and fault-tolerant locking and metadata services—and propose extending existing OSes for disaggregated systems, such as LegoOS [62], with primitives for memory reassignment and failure notification. Below we discuss the proposed primitive operations and the challenges in implementing them, all of which are exacerbated by the fact that the exact nature of disaggregation and the functionality of each component is in flux (§2).

- *Memory grant.* This is a voluntary memory reassignment called by a source application instance to yield its memory pages and move them to a destination application instance. This reassignment requires flexibility from the interconnect or the OS, which must be able to handle modifying memory mappings quickly and at fine granularities.
- *Memory steal.* This is an involuntary reassignment of memory from one application instance to another. While similar to a memory grant from the perspective of the interconnect, a key difference is that the source application instance may not have any prior warning. Since volatile state can now transcend an application instance, the programming model needs to guarantee *crash consistency* to ensure that state is semantically coherent at all times.
- *Failure notification.* An application instance can opt to receive notifications for memory failures or it can register other instances to automatically be notified in such cases. This requires making failure information visible to applications, as well as retaining group membership at the processor so other instances can be notified if the local instance cannot handle or mask the memory failure.

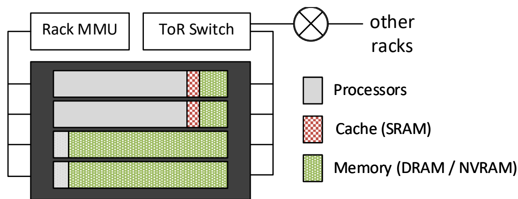

FIGURE 1—Proposed architecture for a DDC. Racks consist of blades housing compute elements (with some local memory) or memory elements (with locally-attached compute to mediate accesses) connected via the Rack MMU. Compute elements within and across racks communicate through a standard Top-of-Rack Ethernet switch.

Data parallel frameworks, such as MapReduce and Dryad, can use these primitives to eliminate unnecessary data transfer during shuffles or between nodes in the data flow graph, while Chubby [9] and other applications based on Paxos [36] can recover and reassign the committed state machine from a failed replica. In addition, early detection of memory failures can trigger recovery mechanisms without waiting for conservative end-to-end timeouts. While this paper focuses on these two applications, we believe that the interfaces are broad enough to benefit other applications. For example, scalable data stores, such as Redis or memcached, could use memory grants to delegate part of their key space to new instances sans copying, while microservice-based applications can use grants and steals to achieve performance comparable to monolithic services and still retain some modularity.

This paper is largely speculative and leaves many open questions. Our hope is to foster a discussion around disaggregation, not from the perspective of operators, but as an opportunity, and a challenge, for systems designers.

2 DDC architecture and resource allocation

In the absence of existing disaggregated data centers, a number of different architectures have been proposed [18, 31, 46, 47, 55, 62]. While these architectures differ in some of the details, the general strokes are similar. We assume the architecture given in Figure 1, which has three core components: individual blades with *compute elements* and *memory elements*, connected over a low-latency *programmable resource interconnect*. While we have chosen to explore these ideas in the context of a single architecture for simplicity, we believe that they are broadly applicable to other disaggregation models.

Compute and memory elements. The basic compute elements in our rack are commodity processors which retain the existing memory hierarchy with private core and shared socket caches. Memory elements, which are conventional DRAM or NVRAM chips, are made available to the compute elements across an interconnect using a low-power local processing element such as a mobile processor or an ASIC. This local processing element interacts with memory through a standard Memory Management Unit (MMU) and interposes on compute element requests to provide addressing, virtualization, and access control; we discuss the trade-offs in offloading more functionality for *near data processing* in Section 7. In line with the majority of proposed architectures, we assume that compute elements have a small amount of locally-attached memory, which is used for the OS and as a small cache to improve performance [21, 62].

Resource interconnect. The resource interconnect allows processor and memory elements to communicate and can be based on RDMA over InfiniBand or Ethernet, Omnipath [8], Gen-Z [22], or a switched PCIe fabric [10, 19]. Our design is agnostic to the physical layer, but we assume a degree of programmability and on-the-fly reconfiguration within the interconnect, which we call the Rack MMU (one can think of it as analogous to an SDN controller and switches). The Rack MMU allows compute and memory elements to be dynamically connected and disconnected in arbitrary configurations. Shoal [63] proposes and implements one such network fabric; although the proposed architecture lacks a programmable switch, it emulates its functionality through a Clos network of switches and a coordination-free scheduling protocol.

Resource partitioning and allocation. We assume that the unit for disaggregation is a single rack (i.e., compute and memory elements reside in the same rack), with resources being partitioned into the desired compute abstractions, such as virtual machines, containers, or processes, and presented to applications (we generically refer to these compute abstractions as processes). The Rack MMU acts as a resource manager for the rack and is responsible for resource partitioning and assigning compute and memory elements to processes.

The Rack MMU has a similar policy for sharing hardware resources as LegoOS [62]: processes may share the same memory element but not the same memory regions (i.e., there is no shared memory). Similarly, compute elements can host multiple processes, but all the threads of a process are restricted to a single compute element. This simplifies caching, as otherwise shared memory would require coherence across the entire rack. Memory is allocated at a page-sized granularity based on the architecture of the compute and memory elements. The Rack MMU is responsible for placement decisions for processes and picks compute and memory elements on the basis of some bin-packing policy, while fine-grained sharing and isolation across co-hosted processes are managed by the local OS.

Addressing and access control. Processes expect to see a continguous private virtual address space, regardless of the physical layout of the underlying memory. To preserve this illusion, the Rack MMU stores a virtual-to-physical (V2P) mapping for each process, which resembles a traditional page table. Compute elements look up these mappings (and may cache them locally) to route accesses to the correct memory element. While we expect the V2P table to be relatively coarse-grained to preserve Rack MMU memory, it needs to support mapping individual pages for memory reassignment.

The Rack MMU is also responsible for configuring access

control to memory. When memory is allocated, the Rack MMU ensures that the topology of the interconnect allows for the existence of a path between the corresponding compute and memory elements. It also configures the page tables at the memory elements with the process identifier (effectively the CR3), the virtual address, and the appropriate permissions, enabling local enforcement at the memory elements.

Scaling out. Not all applications want to live within a single rack: to span racks, traditional Ethernet-based networking is available through a commodity top-of-rack (ToR) switch that connects to the rest of the data center network. Distributed applications comprising multiple processes have to choose the appropriate deployment: intra-rack deployments enjoy lower latencies, while cross-rack deployments have greater failure independence. This decision is analogous to the one faced by developers when selecting the appropriate placement group [4] or availability set [52] in cloud deployments today.

3 Exposing disaggregation

In traditional architectures, the OS is responsible for managing hardware resources, allocating them to processes, and enforcing isolation of shared resources. In a disaggregated environment, this is no longer true and resource allocation is now within the bailiwick of the Rack MMU; the local OS at compute elements continues to be responsible for managing the underlying hardware, providing local scheduling and isolation, and presenting a standard programming interface to applications. Additionally, the OS is responsible for transparently synchronizing application state between local and remote memory and, if any state is locally cached, managing the contents and coherence of this cache [25, 62].

Prior OSes for DDCs [11, 62] have chosen to implement a standard POSIX API and abstract away the disaggregated nature of DDCs from applications. While this allows existing unmodified applications to run on DDCs, we argue—based on our case studies (§5 and §4)—that these applications could achieve better performance if they had more visibility and control. Accordingly, we advocate for the design and implementation of the following operations as OS interfaces.

3.1 Memory reassignment

Memory is reassigned at page granularity by moving it from the V2P mapping of one process to another at the Rack MMU and invalidating any cached V2P mappings at compute elements. Following this, the Rack MMU revokes access to those pages by modifying the page table entries at the source memory element; the detached memory can then be attached to an existing process similar to newly allocated memory. Memory reassignment may be voluntary in the form of *memory grants* or may occur involuntarily through *memory stealing*.

Grants are inspired by L4's *grant* [45], except that they occur in a distributed context; mechanistically, we envision that the source initiates the transfer through a syscall similar to `vmsplice()` with the `SPLICE_F_GIFT` flag in Linux, thereby "gifting" the memory to the kernel and promising to never access it again. After the reassignment at the Rack MMU and memory elements is complete, the source's OS notifies the recipient's OS of the changed page mappings via message passing, which then notifies the destination process via signals.

While grants are the most natural flavor of memory reassignment, they are not particularly useful in the case of compute element failures. An alternative is for other processes to be able to *steal* a crashed process' memory. This is similar to how servers in Frangipani [64] keep their logs remotely, and can request the logs of servers that have crashed to resume their operations. We propose to expose memory stealing via a syscall that requires the id of the source process and a capability; memory allocated using `brk` or `mmap` can disallow reallocation with the appropriate flags.

One questions that arises in both of these cases is who is allowed to trigger memory reassignments and when is it acceptable to do so? While it is clear in the context of grants that a process should have the authority to give away its own memory, the policy around stealing memory is less obvious. One possibility is to group trusted processes together and allow any group member to initiate reassignment; another is to require that a group of processes reach consensus. In such cases, a shared group secret (perhaps based on a threshold secret sharing scheme [61]) may act as the capability to steal memory. We do not enforce a specific policy around reassignment and instead leave it up to the application to determine what is appropriate: while this means that a buggy application can mistakenly steal its own memory and crash, this is not morally different from threads stomping on each other's memory in buggy shared memory applications. We envision memory stealing as primarily an aid to recovery mechanisms when a process has crashed (or is suspected of having crashed), but there might also be cases where stealing memory from a running process is actually profitable.

Maintaining pointer semantics. As reassigned memory pages may contain data structures with internal references, these pages must be attached to the same virtual address to prevent dangling pointers. To avoid a situation in which the receiving process has already used the provided virtual addresses (which would create ambiguity), we propose reserving a fixed number of bits of the virtual address to act as an identifier for the process that allocated the memory pages. As reassigned pages continue to use the same virtual address space, the sender OS marks the virtual address as being "in use" and prevents further allocations or mappings to it.

Crash consistency. Most applications are written with the assumption that application state does not outlive a specific instance and that computation resumes from a clean state after crashes; consequently, they do not maintain their invariants at every point in the *middle* of large operations and have temporary windows of inconsistency. In contrast, memory stealing allows memory to be forcibly acquired at any time or

recovered after a crash—as neither of these scenarios provide the source an opportunity to gracefully make state consistent, remote memory must always be kept crash consistent. Storage systems have historically faced similar challenges and crash consistency for non-volatile memory (NVRAM) is an active area of research [14, 51, 60, 65–68]. Applications can adopt any of these mechanisms, which rely on a combination of techniques such as journaling, soft updates [20], shadow copies [15], and undo logs [14] to remain crash consistent when updating structures in remote memory. Additionally, to verify data integrity after reassignment, applications can use page checksums (or other techniques from persistent storage); the calculation and verification of these checksums can be deferred to memory elements to reduce performance overhead.

Programming models. Crash consistency is necessary, but not sufficient to allow other instances to start operating on objects in stolen or recovered memory: even if consistent, the metadata required to locate the objects may not be available as it is in processor registers, caches, or on the stack. Applications typically rely on the compiler to keep track of internal objects, so when memory is reassigned to a new process, finding the necessary objects from raw memory pages would be a momentous task, akin to searching for a lost treasure.

One possibility is for applications to use an asynchronous, event-based model [2] or a Function-as-a-Service one that forces them to package *all* critical state into an object which persists across invocations. Metadata for these objects (this could be as minimal as the root of a tree) can be stored and distributed in a file system like namespace [17] that acts as a "map" to help locate critical state.

Another challenge is that application developers need to explicitly reason about memory ownership and transfer. While this is a significant departure from existing programming models, there is encouraging precedent: the Rust programming language successfully introduced similar ownership with move semantics into the language itself to guarantee memory safety.

3.2 Failure notification

Compute elements should be notified of memory failures either asynchronously using liveness information from a reliable interconnect or explicitly in response to accesses on unreliable interconnects. In the latter case, compute elements can receive messages from the controller of the memory element (when specific elements have failed), or rely on timeouts (when the entire memory element is unreachable). Error notifications are propagated back to the application through OS signals (`SIGBUS`); applications that want to manage faults can register for these signals and trigger a failure-recovery protocol, while legacy applications may safely ignore them.

As memory failures may result in the loss of application state, it is unclear how an application should leverage failure notifications. One possibility is for the application to preregister a group of processes with the OS that will be informed in case of failures (these processes essentially serve as "emergency contacts"). This group is stored in a per-process forwarding table within the OS; as the OS is local to the compute element, memory failures do *not* affect the forwarding table. This allows other processes to learn of the failure, making the compute element a local failure informer [3, 41].

setting	mean RTT (μs)
Cross-rack (Cloud)	45
Intra-rack (eRPC [28])	2
Future intra-rack (Mellanox ConnectX-6 [1])	1

FIGURE 2—Comparison of the intra and cross-rack latency between VMs. Cross-rack numbers are the experimentally determined mean round-trip time (RTT) between two VMs, guaranteed to be on different racks [4, 52], in a public cloud. As rack co-location is not guaranteed, intra-rack numbers are from the referenced publications.

Failures of compute elements can be detected with a rack-level monitor that periodically verifies the health of compute elements using heartbeats and triggers the appropriate action when failures are detected (e.g., notify emergency contacts of the failure). While the monitor can also fail, it is an optimization, and not a replacement, to failure detection based on end-to-end timeouts. The monitor can set more aggressive timeouts than the application (especially when the application spans multiple racks) because the latency difference between intra-rack and cross-rack is significant, as shown in Figure 2.

4 Case study: Paxos

Applications use Paxos [36] to tolerate failures by replicating their state [35, 56, 59]: Paxos ensures that replicas transition through the same sequence of states. If a replica fails, a client can reissue its requests to other replicas. Failures lead to *reconfiguration*, in which failed replicas are removed and new replicas are introduced to prevent failures from accumulating over time [12, 37, 38]. Reconfiguration brings new replicas up to date by fetching the latest state from other replicas or persistent storage [12], and prevents replicas that have been excluded from the current configuration (presumably because they have failed) from participating if they come back.

Detecting failures. Paxos typically relies on heartbeats with conservative timeouts to ascertain the state of processes. Recent reliable failure detectors [39–41] can quickly initiate recovery mechanisms using local monitors and lethal force. In cases where failures are suspected but cannot be confirmed, these detectors *kill* the replicas—the intuition is that unnecessary failures are preferable to uncertainty.

4.1 Paxos reconfiguration in DDCs

The failure independence of DDCs enables new ways to detect and recover from failures in fault-tolerant applications using Paxos. We assume that replicas run in different racks within the same data center—a reasonable assumption for applications that want greater single-zone failure independence. Within this

deployment, we explore two scenarios: a compute element that loses some or all of its memory elements, and a faulty compute element with functional memory elements.

Dead compute with live memory. When a replica dies, one could reassign its memory, assuming it is in a consistent state, to another compute element and the system could continue operating unimpeded. Such reassignment *reincarnates* the old node, from the perspective of Paxos, allowing the consensus group to return to full health faster (no need to fetch the state from a checkpoint or other replicas).

Should the failure of the compute element be detected faster than the end-to-end timeout of the Paxos group—a likely scenario due to the difference between intra- and cross-rack latencies—reincarnation can be transparent to the rest of the system. In such cases, clients and other replicas only observe a connection termination and will attempt to reconnect. Replicas can register "standbys" with the rack monitor to be contacted when the replica dies and which can take ownership of the dead replica's memory using memory stealing.

In response to the steal operation, the Rack MMU revokes and reassigns access to the region of memory. Revocation is needed because failures are not always fail-stop and the system must prevent a temporarily unavailable compute element from returning and corrupting state. The ToR switch can redirect cross-rack traffic to the new compute element using OpenFlow rules; further, it can also use these rules to fence the old compute element off from the rest of the network [40].

Dead memory with live compute. When a compute element writes to a remote memory element, it is possible for this operation to fail if the memory element is down. Instead of terminating the application right away, as we discuss in Section 3.2, the OS propagates a signal up the stack or forwards the signal to other replicas. This mechanism allows other replicas to detect memory failures and initiate reconfigurations more quickly than relying on end-to-end timeouts.

5 Case study: Data parallel computations

In-memory data parallel frameworks such as data flow and graph processing systems [16, 24, 26, 48, 53, 69] express computations as a series of nodes, where each node performs an operation on its inputs. In these systems, it is often necessary to move data between nodes so that the output of a node may be used as the input to the next node. For example, in MapReduce [16], the output of mappers is shuffled and sent to reducers that operate on a chunk of related data. We believe that executing these systems transparently on a DDC is unwise, and argue that the operators described in Section 3 can cut down on data movement and improve straggler mitigation.

Faster data movement. Deploying a data parallel framework on a transparent DDC results in unnecessary data movement between compute nodes; for example, a single transfer data between two application workers forces 3 network and memory RTTs. First, the source fetches data from its remote memory over the memory interconnect, and then sends it over the network to the destination, who then forwards it to remote memory over the memory interconnect.

Data transfer is often a major bottleneck [50]: Timely Dataflow [49] achieves up to $3\times$ higher throughput when provided with a faster network. Memory grants convert data transfer into a single RTT over the memory interconnect and a control message over the ToR. The source would `grant` the memory pages storing the data that it plans to send to the recipient; the Rack MMU would adjust the necessary page permissions and then notify the source that the grant was successful. The source would then notify the destination that the data pages are ready to be mapped into its local address space.

Once a memory grant is initiated, the contents of memory should not be modified until completed. As discussed in Section 3.1, this immutability is not enforced and it is the application's responsibility to ensure that all its threads have completed before initiating the grant. As data parallel frameworks already have explicit computation and communication phases, we believe this is reasonable; further, any violations are effectively data races and only affect the application itself, but do not impact the broader operation of the system.

Dealing with stragglers. Straggler tasks are often caused by factors local to a particular compute node such as an overloaded processor or insufficient cache capacity or network bandwidth [70]. In such cases, the node can have its memory forcibly reassigned to another node (or set of nodes) by having the job orchestrator `steal` the appropriate memory pages. The recipient node can resume and complete the half-completed computation, rather than starting from scratch. In case of failures, as in Paxos (§4.1), the rack monitor can inform the job orchestrator, allowing it to relaunch the task more quickly; if only the compute elements have failed, the relaunched task can resume computation from where it had stalled.

6 Conclusion

Disaggregation represents a fundamental change in how hardware resources are built, provisioned, and presented to applications for consumption. Early research initiatives have focused largely on building transparent solutions that benefit operators and are application-neutral; for example, LegoOS [62] uses RAID-style [57] memory replication while Carbonari and Beschastnikh propose replication and switch-based failover [11] to preserve existing failure semantics for applications. Carbonari and Beschastnikh also observe that applications could benefit from information about failures but do not go further; we build on that observation and look at how applications that eschew transparency could use this information. More specifically, we borrow ideas from systems for zero-copy IPC [6, 7, 43–45] and RPC [27, 28], distributed shared memory [13, 32, 33, 42, 54, 58], and from reliable failure informers [3, 39–41] for faster recovery.

7 Discussion

We believe disaggregation has merits (and plenty of challenges) and wish to discuss three key areas: firstly, whether the assumptions for our model of disaggregation are reasonable. Secondly, we would like to discuss the role of compute at memory elements and whether there are opportunities to leverage near-data processing (NDP), or if relying on such compute is tantamount to admitting the inadequacy of memory disaggregation. Finally, we would like to discuss feasibility and implementation strategies for the Rack MMU.

Disaggregation model. We assume that the unit of disaggregation will be the rack because it strikes the right balance between increasing provider flexibility around bin-packing resources and the cost and overhead of the (still hypothetical) interconnect. Are there merits to more aggressively disaggregating resources across multiple racks? Further, the interfaces we propose require applications to actively participate in reasoning about disaggregation to benefit—legacy applications, while supported, do not transparently benefit in any way. Is this a reasonable burden for application developers to bear?

Offloading operations to memory. Just as storage systems have moved back and forth between completely disaggregated, disaggregated with offload, and hyperconverged solutions, disaggregated memory systems will need to decide just how powerful and programmable the compute at the memory elements should be. At one extreme, purely disaggregated systems may directly expose memory across the fabric without support for compute—except perhaps very simple functions via processing-in-memory [23]. At the other end, they might be fronted by fully programmable processors running arbitrary code which could adversely affect memory access latencies. In practice, we believe that this is likely to take some form of domain specific language on top of dedicated hardware, or a declarative policy engine on top of a commodity OS and hardware, but with soft realtime guarantees. We believe that this is an interesting area of discussion.

Implementing the Rack MMU. The Rack MMU is assumed to be able to route requests between any compute and memory elements within the rack at very low latency, and to support racks of high density. It is also assumed to have enough space to store address mappings for each process, so that accesses from compute elements are transparently routed to the correct memory element; further, it supports dynamic reconfiguration of routes and mappings without requiring any downtime. In practice, neither of those are completely realistic today: for context, while programmable switches such as the Barefoot Tofino and Cavium XPliant do offer low-latency routing and on-the-fly reconfiguration, they still are limited in their port counts and memory, which restricts their scale. A single switch is unlikely to accommodate the above constraints, so an SDN-like architecture with a slow control plane and a fast data plane could be the sweet spot.

References

[1] Connectx-6 single/dual-port adapter supporting 200Gb/s with VPI. `https://www.mellanox.com/page/products_dyn?product_family=265&mtag=connectx_6_vpi_card`.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2002.

[3] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[4] Amazon. Placement Groups. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html`.

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

[6] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2), 1991.

[8] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*, 2015.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[10] BusinessWire. Liqid Fulfils the Promise of Rack-Scale Composable Infrastructure with General Availability. `https://www.businesswire.com/news/home/20171114006064/en/Liqid-Fulfils-Promise-Rack-Scale-Composable-Infrastructure-General`, 2017.

[11] A. Carbonari and I. Beschastnikh. Tolerating Faults in Disaggregated Datacenters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[12] T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live—An Engineering Perspective. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, 2007.

[13] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-address-space Operating System. *ACM Transactions on Computer Systems (TOCS)*, 12(4), 1994.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through

Byte-addressable, Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

[18] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond processor-centric operating systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[19] S. Foskett. Liqid Takes Composable Infrastructure to a New Level. `https://gestaltit.com/exclusive/stephen/liqid-takes-composable-infrastructure-to-a-new-level/`, 2018.

[20] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2), 2000.

[21] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] Gen-z core specification, revision 1.0. `https://www.genzconsortium.com`.

[23] S. Ghose, A. Borumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-drive perspective. *IBM Journal of Research & Development*, 63(6), 2018.

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[25] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrel, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[27] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[29] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. López-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[30] R. H. Katz. High Performance Network and Channel-Based Storage. Technical Report UCB/CSD-91-650, EECS Department, University of California, Berkeley, Sep 1991.

[31] K. Keeton. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.

[32] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter Technical Conference*, WTEC'94, 1994.

[33] L. Kontothanassis, R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, and M. L. Scott. Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3), 2005.

[34] J. Kyathsandra and E. Dahlen. Intel Rack Scale Architecture Overview. `http://presentations.interop.com/events/las-vegas/2013/free-sessions---keynote-presentations/download/463`, 2013.

[35] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 1978.

[36] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[37] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, 2009.

[38] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1), 2010.

[39] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving Availability in Distributed Systems with Failure Informers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[40] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2015.

[41] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[42] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4), 1989.

[43] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[44] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[45] J. Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9), 1996.

[46] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[47] K. Lim, Y. Turnet, J. Chang, J. Renato Santos, and P. Ranganathan. Disaggregated Memory Benefits for Server Consolidation. Technical Report HPL-2011-31, HP Laboratories, 2011.

[48] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD Conference*, 2010.

[49] F. McSherry. Timely dataflow. `https://github.com/TimelyDataflow/timely-dataflow`.

[50] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics, part 1. `http://www.frankmcsherry.org/pagerank/distributed/performance/2015/07/08/pagerank.html`, 2015.

[51] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[52] Microsoft. Regions and availability for virtual machines in Azure. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/regions-and-availability`.

[53] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[54] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[55] S. Novaković, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[56] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, 1988.

[57] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD Conference*, 1988.

[58] Power, Russell and Li, Jinyang. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[59] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[60] Se Kwon Lee and Jayashree Mohan and Sanidhya Kashyap and Taesoo Kim and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[61] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[62] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[63] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[64] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[65] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[66] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[67] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[68] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[69] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[70] P. Zheng and B. C. Lee. Hound: Causal Learning for Datacenter-scale Straggler Diagnosis. In *Proceedings of the ACM SIGMETRICS Conference*, 2018.