IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 11 February 2023, accepted 25 February 2023, date of publication 28 February 2023, date of current version 3 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3250407

SURVEY

Disaggregated Memory in the Datacenter: A Survey

MOHAMMAD EWAIS, (Graduate Student Member, IEEE),
AND PAUL CHOW, (Life Fellow, IEEE)
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada
Corresponding author: Mohammad Ewais (mewais@ece.utoronto.ca)

This work was supported by an Alibaba Innovative Research Agreement.

ABSTRACT Datacenters of today have maintained the same architecture for decades. The building block of the datacenter remains the server, which tightly couples the necessary compute resources, memory, and storage to run its tasks. However, this traditional approach suffers from under-utilization of its resources, often caused by the over-provisioning of these resources when deploying applications. Datacenter operators allocate the worst-case amount of memory required for each deployed application, which lasts for the entirety of the application's lifetime, even when not actually used. This causes servers to quickly, and falsely, run out of memory before their CPUs have been fully utilized. To address these problems, a new shift in the way datacenters are being built has been gaining more traction. Namely, memory disaggregation. Memory disaggregation can address these problems by decoupling the computational elements from the memory resources, allowing each to be provisioned and utilized separately. While the idea of memory disaggregation is not new, an increasing number of different proposals of memory disaggregation have seen the light in recent years. In this paper, we review many of these recent proposals, and study their architectures, implementations, and requirements. We also categorize them based on their features, and attempt to identify their strengths and shortcomings in an effort to highlight possible directions for future work and provide a reference for the research community.

INDEX TERMS Remote memory, disaggregated memory, memory disaggregation, datacenter architecture.

I. INTRODUCTION

Since the beginning of the Internet, datacenters have witnessed a tremendous increase in their importance and have become an essential tool of our lives. Many of our day-to-day services, such as search engines, social networks, telecommunications, streaming services, and many other personal and business applications, are only enabled by the advancements of today's datacenters. The recent boom of big data and the rising importance and popularity of machine learning and its applications is going to further increase the demand for datacenters. For example, in 2010, datacenters were responsible for 1.5% of the total worldwide energy consumption [1]. This figure grew to 3% in 2017 and is projected to reach 4.5% in 2025 [2].

However, despite years of depending on datacenters, their architecture has remained largely the same. The datacenter has the server as its building block, where each server functions by integrating the necessary compute, network, memory, and storage resources together on a single motherboard. This resource aggregation philosophy has worked well for years, for the datacenter as well as personal computers. However, with Moore's law approaching its end of life, it is becoming exceedingly difficult to expand and scale within the boundaries of a single machine [3]. Furthermore, the inherent difference between CPUs and memories (i.e., the memory capacity wall [4]) and the gap in their performance, power consumption, growth and scaling trends also contributes to this problem, forcing datacenter architects to look elsewhere for solutions.

The problem is further compounded by the inefficient utilization of the resources in today's datacenter. Traditionally,

The associate editor coordinating the review of this manuscript and approving it for publication was Libo Huang.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

applications in the datacenter are deployed in units of virtual machines (VMs) or containers [5]. The amount of CPU and memory resources allocated for each VM or container is decided statically at deployment time and remains constant for its entire lifetime. In reality, the resource consumption of these applications is rarely constant, and varies during its lifetime, as well as depending on the datasets or inputs fed to the application, or other operating conditions like the number of user requests, etc. It is also worth noting that datacenter operators make the allocation decisions based on worst-case scenario requirements, meaning they deploy the application with the largest amount of resources it is expected to use. This is particularly true for memory resources, since running out of memory causes applications to either use swap space, which is notoriously slow compared to memory, or be entirely killed due to lack of resources. Also, CPU resources can be shared, for example, using multi-threading, while memory resources cannot. This worst-case scenario resource allocation, along with the natural variations in application resource consumption, means that for the majority of its lifetime, an application is using less memory resources than what was originally allocated for it. This over-provisioning of resources essentially causes the servers in a datacenter to ''run out'' of memory before their CPUs are fully utilized.

For example; Amazon AWS EC2 had only 7% to 17% CPU utilization [6]. Prior studies have also reported low CPU utilization using Google's datacenter traces, showing a CPU utilization between 28% and 56% [7]. In a more recent study, Alibaba reported 20% to 50% CPU utilization for most of the time, and only as high as 70% at its peak, while at the same time reporting a memory utilization from 80% to 100% [8]. This under-utilization of CPU resources has a multi-fold negative effect. To start with, at least a part of the purchase cost of these compute resources is essentially going to waste, with resources being idle and under-utilized for long periods of its lifetime. Furthermore, this puts a strain on the operation costs of the datacenter, with parts of its energy consumption also being wasted on under-utilized resources. The effects of the wasted energy consumption extend beyond the simple economics of datacenter operation, as they have a negative effect on the environment.

To counter these issues, many recent studies have proposed or revisited the idea of resource disaggregation in the datacenter. This means moving away from a server-centric datacenter towards a datacenter where each type of resource is logically pooled together, completely independent of the other resources. This is already true today in the case of storage, where multiple storage servers exist in a datacenter and can be utilized and shared by many client nodes. Recent proposals extend this idea to the realm of CPU (or compute units in general) and memory, namely, memory disaggregation.

Resource disaggregation allows each type of resource to be provisioned and utilized in fine grained units without affecting other resource types, which further helps to improve resource utilization [76]. However, it also comes with its own set of challenges and problems. For example, connecting pools of different types of resources becomes a much more challenging issue, requiring an upgrade from a chip level (e.g., processor bus or DDR interconnect) or server level (e.g., PCIe) interconnect to a network level interconnect that can provide the necessary high bandwidth and low latency required for efficiently connecting these pools of components together across a rack or even multiples of racks.

In this paper, we survey many of the recent studies targeting memory disaggregation in the datacenter. While we are not the first to survey memory disaggregation techniques [9], prior work was only limited to a handful of somewhat older proposals, thus not representative of the state of the art today. In our survey, we focus on relatively recent proposals, starting with those published after the mid 2000s until the time of writing. We do not cover studies before this time since they were either revisited in the recent proposals or depended on obsolete technologies that render them unusable today. This survey focuses **ONLY** on studies that propose architectures or solutions for memory disaggregation. Studies that focus primarily on improving network bandwidth or latency to help implement memory disaggregation, or studies that focus on file systems for the use of disaggregated Non Volatile Memories (NVMs), or those that focus on rewriting/rearchitecturing applications to fit a memory disaggregation system are **NOT** within the scope of this survey.

For the proposals covered by this survey, we try to classify them based on their high-level architectures, the methods used for connecting clients to servers, their ability to handle data sharing and maintain memory protection, their hardware requirements and software ecosystems, etc. We also attempt to pinpoint the shortcomings of current proposals and use them to provide insight into what we believe should be the direction of future research. However, we do not offer a direct comparison of results between these proposals because of the wide spectrum of different architectures used by these studies, as well as the lack of proper tools (e.g., simulators and benchmarks) built for this purpose. We elaborate more on this in Section V

The rest of the paper is organized as follows: Section II provides a background into many of the current technologies and network protocols that are needed for building disaggregated memory systems. Section III constructs a criteria for comparing memory disaggregation proposals, while Section IV applies this criteria by discussing the proposals, classifying them, and comparing them together to highlight their differences and similarities. Section V lists our observations and recommendations for future work. We then conclude in Section VI.

II. BACKGROUND

In this section we briefly visit relevant topics or concepts that are either constraining factors for memory disaggregation or require modifications or redesigns to support it such as, for example, virtual memory, memory consistency and coherence. We will also examine enablers of memory disaggregation, such as state of the art network protocols with

minimal latency and improved throughput. Finally, we give some examples of applications that can benefit from having a disaggregated memory system.

A. VIRTUAL MEMORY

1) VIRTUAL MEMORY SYSTEMS

Virtual memory is an essential pillar of modern computer systems. It provides each application with the illusion of one contiguous uncontested memory space, simplifying application development and maintaining a uniform view across different machines. The operating system, assisted with the hardware, is then responsible for translating these virtual addresses of the application into physical addresses that can be used to access a machine's local memory. Some of the most important benefits of virtual memory are the following:

- It allows applications to run on any machine, regardless of their available memory resources, without requiring any modifications.
- It allows multiple applications to run on the same machine, without having to worry about conflicts when sharing the underlying memory resources.
- It provides better security due to the isolation of applications.
- It creates the illusion of a larger memory even if the underlying machine does not provide the same amount of physical memory. This also enables using disks as a memory extension through paging, allowing memory pages to be swapped in and out from memory whenever necessary.

2) ADDRESS TRANSLATION

Every memory access in a modern-day processor goes through caching first. From the point of view of virtual memory, there are two ways to design caches. Caches can be virtually tagged, meaning they can be accessed using virtual addresses directly, although a cache miss would still require virtual to physical address translation to propagate through the memory hierarchy. The alternative is to use physically tagged caches which would require virtual to physical address translation prior to accessing the cache. In either of the previous cases, whenever a translation occurs, it first goes through Translation Lookaside Buffers (TLBs), essentially a fully associative cache for the page table, enabling quick virtual to physical translation whenever a TLB hit occurs. In the case of a TLB miss, or misses when there are multiple levels of TLBs, the hardware assists the operating system (OS) by performing what is called a Page Table Walk (PTW), searching through the page table for the virtual address to find a translation. This is a costly operation requiring multiple memory accesses to do a translation.

Even though TLBs do a good job of mitigating much of the latency of address translation, naive PTWs are still costly. This is why modern approaches typically use multi-level page tables. In this scheme, each level of tables contains mappings for the tables of the next level, eventually building a tree

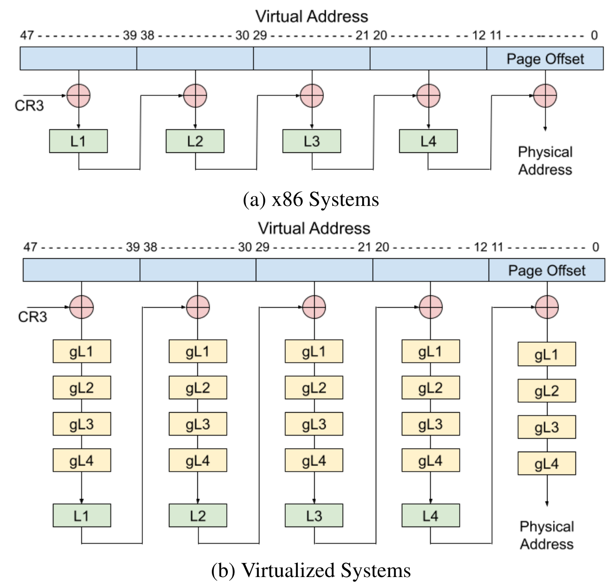

FIGURE 1. Page table walking in x86 and virtualized systems. Adapted from [89].

structure of tables and their entries. A part of the virtual address is used to index each table to generate the address of the next, until the last level table eventually returns the physical address. The first level table is accessed through one of the CPU's control registers. For example, on x86 platforms, the CR3 register [10]. In this scenario, a four-level page table would take four consecutive memory accesses to do one translation as shown in Figure 1(a).

In virtualized environments, hypervisors maintain separate page tables per guest. In this case, a nesting of two page tables is used. One for translating from guest virtual addresses to guest physical addresses, which are still virtual addresses provided by the host. The second translates these addresses to real host physical addresses. Accessing each level of the guest page table has to walk the entirety of the hypervisor maintained table once to generate a physical address of the host page table, incurring 24 accesses in total, as shown in Figure 1(b). To avoid such huge overheads, PTW caches and similarly nested PTW caches were proposed to reduce the number of memory accesses needed [11]. A PTW cache stores entries from all page table levels except for the last, which is cached in TLBs.

B. MEMORY CONSISTENCY AND COHERENCE

1) MEMORY CONSISTENCY MODELS

A memory consistency model is a specification of how the underlying hardware (i.e., CPU) interprets the memory operations in a user program. It is especially important in the context of multi-threaded applications as well as shared memory systems, as it defines the rules of how to maintain a *consistent* view of memory across all processor cores and how memory operations in a processor core will be reordered, if any, and perceived by other cores.

Many memory consistency models exist. We describe some of the most notable ones:

- **Sequential Consistency**: Sequential consistency [12] is the strongest memory consistency model in practice. It behaves as if the memory operations on all cores are executed in some sequential order. It also guarantees that the view of memory operations as seen by different cores is the same as the specified program order. In other words, it does not allow any reordering of memory operations.
- **Total Store Order (TSO)**: TSO [13] allows stores to go through a FIFO-style write buffer, instead of directly to the memory hierarchy. This effectively hides the write latency when stores encounter cache misses, while allowing the processor to continue execution instead of stalling on such writes. This comes at the expense of relaxing some of the sequential consistency, as it allows for stores to be reordered after subsequent loads. Fence operations must be inserted by the programmer whenever needed to handle this inconsistency. TSO is widely used by Intel and AMD processors.
- **Partial Store Order (PSO)**: PSO [14] goes further than TSO by using a non-FIFO write buffer. Essentially, this means that writes inside the buffer can also be reordered as necessary, as they can be committed to memory in a different order than when they were inserted into the buffer. Similarly, fence operations can be inserted by the programmer or the compiler to enforce ordering. PSO was used primarily in SPARC processors.
- **Weak Consistency**: Weak consistency [15] relaxes many of the sequential consistency constraints. It splits memory operations into two categories; data and synchronization (e.g., fences and barriers). Synchronization operations are guaranteed to be seen by all cores in the same order as their program order. Data operations can be reordered freely by the processor, irrelevant from their original program order. However, before a synchronization operation can be performed, all data operations must be completed, and vice versa. There are other variations of weak consistency that have been proposed, like release consistency [16]. It is worth noting that some extreme implementations are completely legal under weak consistency. For example, implementations with multiple concurrent writers are possible. Different variants of weak consistency are in use today by ARMv8 and RISC-V implementations.

TSO is known to be the memory consistency model in Intel and AMD processors [13]. Considering that x86 processors are still the most prevalent in datacenters today [17], that naturally makes TSO the most used memory consistency model. Moving down from TSO to weaker models often allows for better processor optimizations, such as buffering memory operations instead of propagating them directly to the memory hierarchy, allowing processors to proceed while hiding the latency of these memory operations instead of stalling. A side benefit of this buffering is that it allows memory accesses going to the same destination to be grouped together, potentially saving some coherence traffic and allowing for better coherence scalability. Previous studies show that TSO, PSO, and release consistency saw performance increased by 8.9%, 10.6%, and 34.3% respectively over sequential consistency in an 8×8 network on a chip [18]. This decreased load on coherence traffic is particularly important when considering memory disaggregation, as discussed further in Section III. The price paid for this increased performance is that weak consistency models are unintuitive, leaving the programmer to worry about maintaining the consistency and making programming for them a much tougher task.

2) CACHE COHERENCE PROTOCOLS

In today's multi-core processors, each core has its own access to separate local caches. For data to be shared across these cores correctly, these caches must communicate and transfer memory data in between them correctly and coherently. There are two variants for cache coherence protocols, the first is the write-invalidate variant, where a cache line can exist on multiple cores or caches under read permissions, but upon writing, all these copies must be invalidated and only one copy can exist with write permissions. The second variant is the write-propagate, where a cache line can exist in multiple caches, and upon write the data has to be propagated from the writer to all other copies, to maintain coherence. Write-invalidate variants are the most used today, so we only focus on them. The most notable of these cache coherence protocols are:

- **MSI**: MSI is the simplest cache coherence protocol. Under MSI, each cache line can be in one of three possible states. The first such state is *INVALID*, which simply means the line is invalid or the value that exists for it in the cache line is stale. The second state is *SHARED*, which indicates that this cache line exists in read permissions in at least one of the caches, but potentially more. The third state is *MODIFIED*, which means that the cache line exists with write permission in ONLY one cache. A read miss on a cache line results in a request to put the cache line in shared state, which is serviced directly from memory. If the line in question is held in modified state by another cache, it has to be written back to memory and that cache will be downgraded to shared state before the read requester can be serviced. A write miss on a cache line similarly results in the downgrading of all sharers to invalid, including any necessary writebacks to memory if the line is modified, before the requester can be serviced.
- **MESI**: MESI is an addition over MSI. It is based on the realization that many read-modify-write operations will request a cache line in read permissions, then shortly after request write permissions on the same line. To eliminate the need for two requests, an extra state is added, which is *EXCLUSIVE* state. Exclusive state means that a line exists in strictly one cache and only with read permissions. However, when a write miss occurs on a line that is in exclusive state, the cache can upgrade it from exclusive to modified silently, that is without going through the coherence protocol. If a

read miss for the same line occurs in another cache, the owner core will have to be downgraded from exclusive to shared.

- **MOESI**: MOESI further adds on top of MESI to improve performance. In previous protocols, responses to any requests would have to be serviced by memory (or next level cache, if any), which typically has higher latency than a cache in the same level. Based on that premise, a fifth state is added called the *OWNED* state. A cache that has a line in the owned state is one of multiple caches sharing that line, but it exclusively has the right to modify that line. However, it is responsible for broadcasting those changes to all sharers. When a cache line is downgraded from owned statem the modifications have to be written back to memory, and ownership has to be transferred to another sharers, if any. This allows a dirty cache line to be used and moved between caches without hitting the memory, saving up the latency.
- **MOSIF**: MOSIF is highly similar to MOESI and is based on the same premise. It also aims to decrease latency by responding to requests from same level caches rather than memory. Unlike MOESI however, the fifth state in MESIF, called *FORWARD*, can only be used with clean unmodified cache lines. A cache that has a line in forward state must respond to any requests for that cache line.

As the number of states in these protocols grow, more savings in latency as well as traffic on the coherence fabric are achieved, which is extremely attractive for disaggregated memory proposals. However, the cost of these savings are extra complexity on the cache designer, as well as extra memory wasted for the extra metadata bits used to represent the state. Whether this cost is justified should be decided on a case-by-case basis. For more information regarding the above coherence protocols, please refer to [19].

While the discussion above focuses on cache coherence primarily, the same concepts apply to memory coherence. For example, when a distributed shared memory is used by multiple processors, and copies of pages exist on more than one of these memories, the same protocols may be used for such cases. A popular example for memory coherence is the Ivy protocol [20], which is essentially an MSI memory coherence protocol. The mechanism used to implement these coherence protocols is usually one of the two following:

- **Snooping Based Coherence**: In snooping based coherence [21], all the caches share a communications bus. Each cache will monitor (i.e., snoop) all transactions on the bus, filter the transactions to what is included in the cache, then update itself based on these transactions if necessary. Snooping based coherence is fast, as it does necessitate going through a central structure. However, it suffers from scalability issues. As the number of caches on the bus grow, so does the requirements for bus bandwidth and latency.
- **Directory Based Coherence**: Directory based coherence [22] tries to fix the scalability issues of snooping based coherence. In directory based coherence, a central structure called the directory is used to manage the state of the caches instead of a central bus. The directory receives requests from all caches, and is responsible for forwarding these requests to the necessary caches only. Because it does not require broadcasting like snooping based coherence, directory based coherence can better scale up and is used in many recent designs.

C. NETWORKING

1) COMPUTE EXPRESS LINK (CXL)

Compute Express Link (CXL) [23] is a PCI Express based open industry standard for high speed interconnects. It allows connection between multiple CPUs, accelerators, IO devices, and even remote memories in a cache coherent manner. The CXL standard has encompassed other existing standards such as OpenCAPI, Gen-Z, and CCIX. The CXL standard also introduces the idea of a CXL switch that can enable fanning out to multiple devices. CXL has not reached commercialization yet, and thus almost no proposals discussed in this survey depends on it. But, it is anticipated to help improve memory utilization for disaggregated memory systems.

2) REMOTE DIRECT MEMORY ACCESS (RDMA)

Remote Direct Memory Access (RDMA) is a user-level network protocol that allows direct memory access over network. It provides a zero-copy approach to data movement as it allows data to be transferred directly through the network adapter without copying it to OS data buffers first. RDMA communicators use queue pairs (QPs) consisting of a send queue and a receive queue. Processes access QPs using RDMA *verbs*, which can be one sided or two sided. One sided verbs like *read*, *write*, and *atomic* require no involvement from the receiver CPU and can completely bypass it and directly access its memory. Two sided verbs like *send* and *receive* require the remote CPU to be involved. For instance, a receive request asks the receiver for the address of a buffer to be used by a subsequent send request.

RDMA can be realized in three modes:

- **Reliable Connection**: Reliable Connection (RC) RDMA requires a connection to be established between two points. Each connection uses a separate QP, which suffers from scalability issues as more connections can result in too many QPs for the Network Interface Card (NICs) to handle, eventually causing cache thrashing in the NIC. As the name suggests, RC RDMA ensures data reliability and ordering in the network layer, and can return errors in cases of failure. RC supports all one sided and two sided verbs.
- **Unreliable Connection**: Unreliable Connection (UC) RDMA also requires a connection to be established between two points. Thus, it suffers from the same scalability issues as RC. However, UC provides no

reliability guarantees, but can provide better throughput as it saves the bandwidth needed for acknowledgment packets. UC supports two sided verbs, as well as the write variants of one sided verbs only.
- **Unreliable Datagram**: Unreliable Datagram (UD) RDMA is connectionless. It does not require a connection to be established from point to point, which allows it to use one QP to communicate with multiple destinations, which, in turn, gives UD better scalability characteristics. Like UC, UD also provides no reliability guarantees. UD supports only two sided verbs.

There are currently several different implementations of RDMA. The most common of which are Infiniband, RDMA over Converged Ethernet (RoCE, RoCEv2), and internet Wide Area RDMA Protocol (iWARP). They are the basis of many of the proposals discussed in this survey. There are several studies that aim at improving the performance of RDMA or proposing new RDMA verbs for certain applications, these are not the scope of our paper and are not discussed.

D. APPLICATIONS OF DISAGGREGATED MEMORY

1) DATABASES

Historically, databases have utilized secondary storage, such as disks, as the primary backing store for their tables. With the rise in access latencies associated with external storage, databases have increasingly migrated towards utilizing main memory as their backing store [25], [26] or, at minimum, as a cache for their tables. This shift towards in-memory database systems is driven by the growing size of database tables, making databases a naturally memory hungry application. Memory disaggregation can directly address these requirements by providing the necessary memory resources to databases, improving their scalability and performance. As a result, some of the proposals we cover in this survey (e.g., [90], [93]) focus on building systems that directly enhance the performance of databases.

2) GRAPH APPLICATIONS

Graph applications are another type of inherently memory hungry applications. Graphs require large amounts of memory to store their edges and vertices. With larger graphs, the memory requirements can quickly and easily deplete the memory of a single node. Disaggregated memory can thus be a potential solution for the memory problem of graph applications by decoupling compute from memory and thus allowing more memory resources to be used for graph applications as needed. Though none of the articles described in this survey are directly aimed at graph applications, graph applications in the context of memory disaggregation has been studied before [27].

3) MACHINE LEARNING

Like databases and graph applications, machine learning has also seen an increase in its memory footprint, with larger and larger models being trained everyday (e.g., the infamous ChatGPT [28]). Machine learning can benefit from memory disaggregation in the same way, i.e., by expanding the memory resources for it allowing it to utilize the amount of memory it needs rather than be restricted by a single node's worth of memory. This would directly result in better training times since memory disaggregation would eliminate the need for data transfers between memory and disks. At least one proposal in this survey is focused on building memory disaggregation systems specifically for deep learning [78].

4) VIRTUAL ENVIRONMENTS

Since virtualization was the root cause of the resource over-provisioning discussed in Section I, it is only natural that virtualization stands to benefit from memory disaggregation. By decoupling compute resources from memory resources, a virtual machine can request and utilize the exact amount of memory it needs at any given point in time. Not only that, but if the underlying disaggregated memory system supports data sharing, a virtual machine can also extend beyond using one physical machine. In fact, the latter solution is discussed in some of the proposals covered by this survey [48], [49], [50], [51].

III. TAXONOMY

To be able to classify state of the art proposals of memory disaggregation, we have to establish a clear taxonomy to facilitate this task. We classify proposals based on the following traits and parameters; system architecture, client and server architecture, support for memory sharing and memory protection, along with memory consistency and coherence, when available, and finally, fault tolerance. In each of the following subsections, we focus on one specific aspect of memory disaggregation and outline a classification criteria for later use.

A. SYSTEM ARCHITECTURE

In the proposals discussed by this survey, there are two main methods to build disaggregated memory systems, as shown in Figure 2. These are the following:

- **Split**: In this scheme, the memory attached to each node is logically partitioned. The first part of which is strictly only used by its owning node, while the second part may be advertised to, and utilized by other nodes in the cluster. The partitioning may be fixed or variable. That is, the size of the partitions can be defined at design-time or vary at run-time depending on demand. In this scheme, a node can be a client and a server at the same time. In other words, it can be accessing remote memory while at the same time serving some of its own local memory to be used by other nodes. By definition then, this scheme requires two components. The first is a daemon that runs on each node, receiving and handling requests for memory from other nodes. The second is a client running on nodes that request memory. These can be implemented in OS, hypervisor, or user-space as discussed later, but cannot, at least trivially,

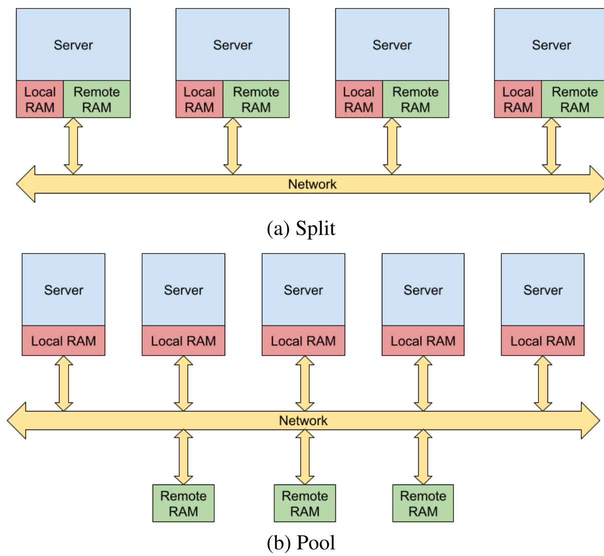

FIGURE 2. High-level system architectures for memory disaggregation.

be implemented in hardware. This scheme does not require any restrictions to other design aspects such as data sharing, memory protection, consistency, and coherence.

- **Pool**: In this scheme, each node would contain its own local memory that is not to be shared with other nodes. However, there also exists a separate pool of memories to be used by these nodes. Unlike the previous scheme, this scheme provides clear separation between compute and memory domains, since it uses separate nodes for both. Thus, it can potentially free compute nodes from the burden of running a memory server side by side with computations. However, it is also possible to blur this separation again by introducing near memory processing on the memory nodes, although very rarely realized in the proposals we discuss. Unlike the first scheme, this is completely orthogonal to all other design aspects, including the implementation of client and server nodes, as well as sharing, consistency, and coherence.

B. CLIENT IMPLEMENTATION

There are many ways to implement a client that can function in and utilize a memory disaggregation system. In the following we discuss the ways most commonly used in previous studies to implement clients:

- **User-space Application**: In this case, the memory disaggregation is explicit, and the programmer is required to understand the underlying memory system and code using a specific library/API to allocate, free, share, transfer, and protect data as necessary. Proposals that opt for this design argue that no changes to the hardware or OS allow for easier and quicker adoption of their designs. However, it also means any target applications have to be rewritten to utilize these APIs and make use of the disaggregation. Often times, these proposals, if they support data sharing, also involve some form of weak memory consistency models, making it even harder on the programmer to rewrite such applications. This approach is one of the most common in the proposals we survey.
- **Hypervisor**: Proposals that use this level of implementation attempt to hide the memory disaggregation from the user through modifying hypervisors. This allows applications to run on such systems unmodified, and oblivious to the traits of the underlying memory system. It also allows new applications to be developed with ease and without requiring new knowledge. The drawback is that it requires modifications or potentially a complete redesign of the hypervisor, especially if data sharing is supported in such implementations, since it also requires extra code to handle consistency and coherence. Since memory disaggregation is implemented at the hypervisor level, an obvious limitation is that this method cannot be used for native OSes or other types of virtualization (e.g., Containers).
- **OS**: OS implementations of a memory disaggregation client are quite diverse, and can take shape in a multitude of forms:
  - *Disk*: In this case, the remote memory appears to the OS as a network backed disk. This only requires a device driver to be written, but no modifications to the OS kernel itself. Because the OS sees the remote memory as a disk rather than a memory, data sharing and subsequent coherence and consistency are not possible. Furthermore, it is the responsibility of the user to explicitly map the remote memory disk to memory (e.g., *mmap* [24] it, on Linux systems) to be able to utilize it in their application.
  - *Swap Space*: This builds upon the previous case by using the remote memory disk as a swap space for the OS. Since swapping is always hidden from the programmer, it does not require any extra work on the programmer's part. A pre-compiled application can simply run on such a node, and once it starts running out of local memory, it will be up to the OS to swap pages between local and remote memory as necessary, much like how ordinary swap spaces are used today, possibly with a lower latency, depending on the latency of the remote memory and its remote connection. This also utilizes existing OS capabilities, like page prefetching, with no modifications to the kernel. However, like its predecessor, this still suffers from the inability to share data between different clients. This is one of the most common approaches found in our studies.
  - *Page Fault Handling*: This is an upgrade built upon the swap space solution, which offers one important advantage. By using page fault handlers, it allows the freedom of disaggregating any part of the memory at any time, as opposed to swap which only allows doing so when local memory runs out.

  - *NUMA*: This approach is fundamentally different from using *Swap Space* or *Page Fault Handling* since it uses remote memory as a memory rather than a disk. It exposes the remote memory as part of a Non Uniform Memory Access (NUMA) space, which can then be used by the kernel to satisfy application requests. In this case, the OS would try to satisfy allocation requests from local memory first, before resorting to using remote memory. It is also the OS's responsibility to handle page migration between the two sections of memory. This approach requires that the underlying hardware be already coherent, or that major modifications be introduced to the kernel itself to support non-coherent remote memory. The latter option is not actually realized in any of our studied proposals. It is worth noting that allowing data sharing in such implementations requires an even larger kernel modification to be able to communicate with other nodes, which is potentially why this is also never proposed in our studies.
- **Hardware**: Hardware implementations of memory disaggregation are by far the most complex to implement. However, they provide an implicit approach to implementing memory disaggregation with support for legacy applications with no modifications. Hardware implementations discussed in this survey usually expand upon existing memory hierarchies of modern CPUs. This is achieved by simply introducing new network enabled remote memory controllers, which enjoy sharing the processor's coherence infrastructure. Because they are based on the same memory hierarchy design concepts, it is also possible to utilize existing ideas and implementations such as hardware-based prefetching, as well as caching remote memory through existing processor caches. For research purposes, such designs can be realized using simulators substituting for ASIC design, or through cache coherent FPGAs closely attached to processors.

These are the most common approaches used for building clients in a disaggregated memory environment. They serve as umbrella categories under which we can classify most proposals. Some proposals use combinations of these approaches, while some others exist completely outside these umbrellas, we discuss their approaches when we discuss the classification itself.

C. SERVER IMPLEMENTATION

Server implementations can more or less be categorized under the same umbrellas as client implementations, with notable additions, modifications, and exceptions as we discuss next. Server implementations also need not be coupled with their client counterparts. In other words, clients and servers can be implemented in different ways using different methods. The most common approaches for implementing memory servers are the following:

- **Software**: This is usually implemented as a simple daemon-like program that is responsible for mainly two tasks. The first is to receive network requests for memory and respond to them. The second task is to allocate such memory from the OS of the node it is running on. The order of these operations may be reversed, i.e., the daemon may allocate a memory pool first then start servicing requests by allocating from it. Depending on whether or not the system supports data sharing, it may also do a third task, namely responding and handling memory coherence requests. The level at which the application exists varies, with some proposals putting it as a user-space daemon, while others implementing it as a kernel-level module or daemon.
- **Key-Value Store**: This is a specialized form of a user-space application, where the application essentially presents itself as a key-value store for its clients. Unlike other options discussed here, this is very restrictive, as it only works with user-space based clients that utilize a library/API with predefined data structures that can use the remote memory key-value store. Key-value stores can provide data sharing and atomicity, as discussed later.
- **Hardware**: Hardware implementations of memory servers yield what is called *memory blades*. Essentially, a component that includes the necessary memory, whether DRAM or non-volatile memory based, along with a network capable memory controller. These servers can only be used to access memory, as they do not host any compute components. Since these are expensive to build as ASICs, researchers often substitute them by using simulators or emulators. It is theoretically possible to implement these through FPGAs, but none of the current studies, to the best of our knowledge, provide such implementations.

As with client categories, these are the server implementation categories that can classify most proposals. During classification, whenever any of our studied proposals fall outside these categories, we discuss their approaches as we present them.

D. ADDRESS SPACE AND TRANSLATION

A natural aspect to consider with memory disaggregation is the virtual and physical address spaces used by running processes, and how to manage virtual to physical address translations. The following options exist in our studied proposals:

- **No Change**: This happens especially in virtual memory swap based systems. Since swapping naturally operates behind the scenes and is hidden from the application, virtual memory addressing remains the same with no changes. Translation is also the same, going through TLBs then page tables with no modifications. Some other non-swap based systems also follow this approach, especially those that maintain support for legacy applications.

- **Partitioned Global Address Space (PGAS)**: In this scheme the address space is partitioned into two main parts. The first part is local to each process, while the second is global and shared between all of them. Translation can still occur in the same normal procedure through TLBs and page tables, but would require special page fault handling to map local pages to the same node and handle global pages accordingly.
- **Unified Address Space**: In this case, the entire memory disaggregation system is available under one virtual address space. The physical address is either flat, or partitioned to address memory nodes followed by addressing memory at each node. Translation can be complicated in this case, since it is essentially extended beyond local memory. It requires global page tables, along with possible page table caching. The disaggregation system, if hardware or OS based, would also have to handle different core TLB shootdowns (A form of coherence where TLB entries are discarded when they go out of sync) to maintain TLB coherence across compute nodes.
- **Key-Value Store**: This is not a typical addressing mode, but it is used in systems that utilize a key-value store as its backend. Key-value stores as the name suggests uses keys to access data, essentially like a dictionary. There is no translation required for such addressing. The address is sent to the key-value store, which responds back with the data structure stored at this key.

E. MEMORY TRANSFERS

Memory transfers in memory disaggregation systems can be categorized in two different ways. The first is based on the network protocol used to transfer the data. Theoretically, any type of interconnect, network or otherwise, could serve as the infrastructure for data transfer in memory disaggregation systems. Protocols like TCP/IP and UDP/IP are used in older proposals, but they are far from offering the bandwidth and latency requirements for memory disaggregation today. Furthermore, they also introduce the extra latency of going through the OS stack before actually reaching the network.

PCIe-based protocols like CXL [23] (or any of its predecessors) are good candidates for use with memory disaggregation. By design, these protocols support cache coherence, which makes them suitable for hardware-based implementations of memory disaggregation. However, they are limited to short distances only and cannot expand beyond a single rack slot or at most an entire rack. RDMA-based communication is, at least currently, the most used protocol for memory disaggregation. RDMA is also suitable for memory disaggregation since it avoids the OS stack as well as the CPU on the destination. Some proposals opt for designing custom network protocols, as we will discuss in SectionIV.

The second way to categorize memory transfers is based on the granularity of data being transferred itself, which in turn is highly dependent on the choice of client and server implementations. Memory disaggregation proposals that use a key-value store as its memory server have memory transfers that vary in size depending on the data structure being transferred itself. On the other hand, proposals that depend on virtual memory, like OS-based swapping or page faulting, naturally use memory transfers that are page sized. If data sharing is enabled, these proposals would usually suffer from an issue called *dirty data amplification*, which happens when an entire page is marked dirty even when only a small part of it truly is. This impacts the network traffic used for coherence, amplifying it unnecessarily. To alleviate this problem, other studies propose cache line granularity for data transfers, which would decrease, if not eliminate, dirty data amplification. However, because the amount of data sent on each request is much lower, this could potentially come at the cost of extra requests and potentially performance degradation.

F. MEMORY CONSISTENCY AND COHERENCE

Many of the proposals discussed in this survey do not support data sharing at all. In other words, in these implementations, each remote memory page can only be used by at most one client. This means that for these proposals, there is no requirement for memory consistency or coherence, other than the guarantee of correct order of requests.

For the remainder of our studied proposals, they span the entire spectrum of consistency and coherence solutions. For example, many software-based proposals opt for a weak consistency model. Since software proposals do not honor legacy code and require programmers to rewrite their programs, it is also possible to resort to a weak consistency model at no additional cost. It is not uncommon to see some of these proposals use a multiple-reader multiple-writer implementation for such weak consistencies. On the other hand, proposals that maintain backward compatibility typically choose the same memory consistency model as the compute units it uses, which is typically x86-TSO. Some proposals opt for complete sequential consistency, which has no dependency over the existing hardware since the semantics of sequential consistency are legal under all weaker consistency models.

On the coherence front, many proposals also prefer to avoid coherence altogether, or at least make it the programmer's responsibility to communicate coherence whenever necessary. This approach relieves a lot of the network requirements needed for coherence. Other proposals either extend the same coherence protocols used in their processors, or use known implementations like the Ivy protocol and build upon them to suit the scale of disaggregated memories.

G. CACHING AND PREFETCHING

Caching and prefetching are not essential when it comes to disaggregated memory architectures, but they can be utilized to improve the design significantly by improving access latencies. Prefetching in our studied proposals is restricted to virtual memory based implementations, and is exclusively realized through OS page prefetching or speculative page faults. Prefetching pages eliminates the extra latency of page faults and hides the latency of fetching

these pages off the critical path. Caching is also limited to two general methods of implementation. Naturally, software implementations provide caching in their corresponding software/library components. Some implementations use the local DRAM of their client nodes as a cache, either in part or in full, with some proposals even using hardware to manage the DRAM cache. To our knowledge, no studies have proposed extra levels of caching, i.e., caching between client main memory and remote memory servers.

H. FAULT TOLERANCE

Fault tolerance is essential in datacenters. Failures of nodes are a fairly common event, and must be handled properly. Furthermore, mass failures due to power outages or other reasons are also not uncommon. If fault tolerance is not supported or enabled, the results of such outages would be devastating. When discussing fault tolerance, we mainly focus on fault tolerance for the memory itself. For example, network fault tolerance, including retries, latency guarantees, and other techniques, are not covered by this survey. There are multiple ways fault tolerance is enforced in the memory disaggregation proposals:

- **Replication**: Replication may be the easiest as well as the most straightforward option to support fault tolerance. In essence, every memory unit has multiple concurrently existing copies in physically different locations or memory devices. Any writes have to be propagated to all copies, while only one is needed to serve read requests. The drawbacks of replication are obvious, the price paid in redundancy to achieve replication is expensive in terms of used resources. This becomes even more severe when a higher number of replicas is required to achieve better fault tolerance.
- **Logging**: Logging is somewhat similar to replication, although it can be thought of as more fine grained. Every operation that results in the modification of data or bringing in new data is logged in a secondary (or more) location. In the case of failure, such logs may need to be traversed so that a recovery of data may be constructed.
- **Persistence**: Persistence is achieved in two ways. The first is simply replicating memory contents to a storage device (i.e., disk). Because of the performance gap between memory and storage, this type of persistence can cause tremendous slowdowns, so most proposals opt to do it off the critical path. The second way utilizes non-volatile memories, and thus requires no replication to achieve persistence.
- **Erasure Coding**: Erasure coding aims to achieve similar fault tolerance to replication, as well as mitigate its excessively high cost. Erasure coding is essentially an error correction coding. It encodes data of some length into a redundant format with a higher length. This longer encoding is then split across multiple memory devices or locations. The original format of data can be recovered from a subset of this longer encoding, essentially that means a system can tolerate to lose some of these splits without an issue. Increasing the length of the encoding ensures better fault tolerance, at the expense of more memory resources. The drawbacks of erasure coding are higher memory access latencies, as they include the time needed for encoding and decoding the data. Another drawback is higher strain on the network, since a single data access requires multiple separate memory access from multiple locations.

IV. CLASSIFICATION

In this section, we apply our taxonomy to existing memory disaggregation studies. We briefly visit each of these proposals, discuss their implementations, compare them, and show their similarities and differences. The section is organized in four subsections. The first covers high-level system architecture. The second subsection focuses on memory hierarchy. While the third subsection covers fault tolerance. We also add an extra subsection for discussion studies that do not provide implementations and do not fit the aforementioned classification.

A. ARCHITECTURE COMPARISON

In this subsection we discuss the general system architecture, client and server implementations, as well as data transfers. These are the entry point to comparing memory disaggregation proposals. We leave other details for the next subsections. Tables 1A and 1B summarize the comparisons of this subsection.

1) OS BASED SYSTEMS

In this subsection, we cover proposals that mainly depend on OS features or even require OS kernel modifications to support memory disaggregation. We categorize these proposals based on their required or used OS features:

a: SWAP

MemX [29] is the earliest proposal we include in our survey. The paper presents a pool system architecture, although its components can be, in theory, implemented in a split architecture too. MemX clients see the remote memory as a block device (i.e., disk) that is directly used as a swap space. This is implemented either through Kernel modules that can be used in guest OSes or the host OS directly, or using a hypervisor driver that is shared by all guest OSes. MemX servers use a separate network capable kernel module running on a native OS. Since MemX is swapping based, it maintains a memory transfer unit size of 4KB. MemX uses a custom protocol, called the remote memory access protocol, which bypasses the TCP/IP stack and communicates directly with the network device. The MemX system also does not support data sharing. A highly similar approach is that introduced by collaborative memories [30] as it also uses the remote memory as a network connected block device. Unlike MemX, it makes use of the existing TCP/IP stack to transfer data, in an effort to avoid OS as well as application modifications.

Infiniswap [31] introduces a detailed swapping system to be utilized for memory disaggregation. It essentially uses remote memory as a cache for disk swapping. Like

TABLE 1. Architectures and implementations of memory disaggregation proposals.

Paper	System Arch.	Client Implementation	Server Implementation	Transfer Granularity	Transfer Protocol	Data Sharing
MemX [29]	Any	Swap Space	Kernel Module	4KB	Custom (RMAP)	
zswap [35]	Any	Swap Space	User-space App	4KB	Unknown Network	
Infiniswap [31]	Any	Swap Space	User-space App	1GB	RDMA	
FastSwap [32], [33]	Split	Swap Space	Kernel Module	4KB	RDMA	
Valet [37]	Split	Swap Space	Kernel Module	4KB	RDMA	
XMemPod [34]	Split	Swap Space	Kernel Module	4KB	RDMA	✓
CFM [36]	Split	Swap Space	User-space App	4KB	RDMA	
Leap [46]	Split	Swap Space	User-space App	4KB	RDMA	
Collab. Mems [30]	Pool	Swap Space	User-space App	4KB	TCP	
Canvas [39]	Pool	Swap Space	User-space App	4KB	RDMA	
SemSwap [38]	Pool	Swap Space	Kernel Module	4KB	RDMA	
FluidMem [40], [41]	Pool	Page Fault Handler	KV Store	4KB	RDMA	
Hotpot [43]	Split	Kernel Module	Kernel Module	4KB	Custom (RDMA)	✓
Remote Regions [44]	Split	Kernel Module + Software Library	User-space App	128KB	RDMA	✓
DeX [45]	Split	Kernel Modification	Kernel Modification	4KB	RDMA	✓
Hydra [47]	Split	Swap Space + Kernel Module	User-space App	4KB	RDMA	
Hydra [47]	Split	Kernel Module + Software Library	User-space App	4KB	RDMA	
vNUMA [48]	Split	Multi-node Hypervisor	Multi-node Hypervisor	4KB	Custom (Eth)	✓
GiantVM [50], [51]	Split	Multi-node Hypervisor	Multi-node Hypervisor	4KB or 4MB	RDMA	✓
vDSM [49]	Split	Hypervisor Swapping	User-space App	4KB	RDMA	✓
Semeru [52]	Pool	Multi-node JVM + Swap Space	Light JVM	4KB	RDMA	
MemLiner [54]	Pool	Multi-node JVM + Swap Space	Light JVM	4KB	RDMA	
Mako [55]	Pool	Multi-node JVM + Swap Space	Light JVM	4KB	RDMA	
DLM [56]	Pool	Software Library	User-space App	User Defined	TCP	
FaRM [57]	Split	Software Library	User-space App	64B to 1MB	RDMA	✓
GAM [58]	Split	Software Library	User-space App	512B	RDMA	✓
At Scale [59]	Split	Software Library	User-space App	4KB	RDMA	
Grappa [60], [61]	Split	Software Library	User-space App	Data Structure	RDMA	✓
AIFM [66]	Split	Software Library	User-space App	Data Structure	DPDK (TCP)	
rMap [64]	Split	Software Library	User-space App	4KB Multiples	RDMA or TCP	
Argo [62]	Split	MPI-based Software Library	User-space App	4KB	TCP	✓
MAGI [63]	Split	Kernel Module + Software Library	User-space App	4KB	RDMA	✓
CODS [65]	Split	Kernel Module + Software Library	Kernel Module	Unknown	PCIe	

TABLE 1. *(Continued.)* **Architectures and implementations of memory disaggregation proposals.**

Paper	System Arch.	Client Implementation	Server Implementation	Transfer Granularity	Transfer Protocol	Data Sharing
Clover [68], [69]	Pool	KV Client	KV Store + Programm. Switch	Data Structure	RDMA	✓
Concordia [70]	Any	User-space App	User-space App + Programm. Switch	4KB		✓
MIND [71]	Pool	Kernel Modification	Kernel Module + Programm. Switch		TCP + RDMA	✓
blade [72], [73]		Hypervisor Swapping	Memory Blades		PCIe	
		CPU Blades		Cache Line	Coherence Inter.	
dReDBox [74], [75] , [76]				N/A	N/A	
DirectCXL [77]				Any	CXL	
MC-DLA [78]		CPU/GPU Blades		Unknown	NVLINK	
Clio [80]		Software Library	Memory Blades + PIM	Data Structure	Eth	✓
LegoOS [81]		CPU Blades	Memory Blades	4KB	RDMA	
					Eth	
PBerry [82]	Pool	ccFPGA + Kernel Module	N/A	Cache Line	RDMA	
Kona [83]		ccFPGA + User-space App				
ThymesisFlow [84]			ccFPGA + User-space App		OpenCAPI + Aurora	
soNUMA [88]	Any	RDMA-like NIC + Software Library	RDMA-like NIC + Software Library		Custom	
DeACT [89]	Pool	CPU Node + Translator Module	Memory Nodes + Central Translation Unit	4KB	Simulated	
MEMSCALE [90]	Split	CPU Node + Memory Controller	Memory Nodes	Cache Line	HyperTransport	
DVM [92]		ISA (Emulated)	ISA (Emulated)	4KB	Unknown Network	✓
Farview [93]	Pool	Software Library	DPR FPGA	Data Structure	RDMA	
Network Supp. [94]		Swap Space	Emulated	4KB	N/A	
Network Reqs [95]						

MemX [29], it is configured as a block device used as a swap space. However, swaps to remote memory are also forwarded to storage devices off the critical path. Infiniswap uses units of 1GB slabs each, and sends data using RDMA. The paper covers more details about placement, eviction, batching, as well as remote memory reclamation. Another similar proposal is FastSwap [32] which tries to mitigate the drawbacks of Infiniswap. Infiniswap does not allow multiple clients to use the same block device, even when there is unused memory. FastSwap fixes this issue by dynamically sharing memory between multiple hosts, although it is limited to VM clients on a single host. It also introduces compression on swap outs and decompression on swap ins to save the amount of memory utilized. In a subsequent paper [33], FastSwap is incorporated into a larger memory disaggregation system where it is extended to use remote memory over RDMA. XMemPod [34] was also introduced with similar compression features. It provides two levels of remote memory to client VMs, the first being the host memory, and the second being outside-host remote memory. It also supports data sharing between different VMs on the same node, but not beyond.

Another swap engine is Google's zswap [35], which also supports data compression and movement across a network, although no specific network protocol is specified. The novelty of zswap is that it identifies cold pages (infrequently used pages) and proactively compresses and swaps them out, as opposed to doing so on page faults. This saves memory space as well as decreases the page fault latency by not evicting on the critical path. CFM [36] also introduces another swapping system with active memory swapping out similar to zswap, although it does not compress its data. CFM also separates the swapping in of requested and prefetched pages, and assigns them different priorities to improve page

fault response time, and uses RDMA for data transfers. It is worth noting that CFM's swapping system is also called FastSwap, which is not the same as our previously reported FastSwap [32]. Valet [37] is another remote memory swapping engine. It provides multiple improvements on the normal Linux page swapper. First, it uses a local page cache for remote pages. It pre-allocates pages in this cache and uses these pages when needed, cutting down the access time considerably. It leaves writes to happen off the critical path for the same reason. Towards the same end, user freed pages are put pack into the local cache for later re-use instead of actually freeing them. Like many similar proposals, Valet uses RDMA for its 4KB sized data transfers.

The two final swap engines we visit have somewhat different focuses. SemSwap [38] is focused on reducing the effects of dirty data amplification. Unlike normal swap engines, SemSwap does not view its applications as black boxes, it instead collects information from its JVM runtime about the hotness of data lines within virtual memory pages, then consolidates some of these hot parts together in a single physical page, thus reducing the amount of swapping needed. SemSwap guarantees correctness and can revert to the original physical pages whenever a cold data part is needed. The second final swap engine is Canvas [39] which is instead focused on reducing the interference caused by multiple memory client applications competing for swap space, locks, as well as RDMA resources. Canvas introduces application private swap spaces as well as private per-application RDMA bandwidth. It also maintains a global swap space for shared pages. Canvas also introduces some extra optimizations enabled by the isolated swap spaces such as adaptive scheduling and prefetching techniques.

b: PAGE FAULT HANDLING

FluidMem [40], [41] uses a different approach, although still based on swapping. Instead of introducing the remote memory as a block device, it uses a custom page fault handler to deal with remote memory swapping on its own. FluidMem also differs in its server architecture, where it uses a normal server hosting RAMCloud [42], an in memory database, as a backend key-value store, and communicates with it over Infiniband RDMA.

c: FILE-LIKE

Another different approach is that proposed by Hotpot [43]. Hotpot makes use of non-volatile memories, and tries to blur the line between memory and storage. Hotpot uses file-like structures to name data in its memories, and provides a kernel module with an *mmap*-like functionality to allow allocating and naming such data. The pointers resulting from these mmap-like calls are used similarly to normal pointers without any extra function calls. These pointers are guaranteed to be consistent across machines as well as persistent across application reruns. Hotpot also supports data sharing on a 4KB page granularity and over an RDMA interface. Similar to Hotpot, Remote regions [44] also follows the same approach of using regions (i.e., files) as an abstraction of remote memory, although it is restricted to volatile DRAM memories. It also provides a file system kernel module to enable this abstraction, as well as a software user library to allow data allocation, freeing and sharing. Remote regions uses 128KB chunks over RDMA.

d: THREAD MIGRATION

DeX [45] proposes to completely reverse the solutions, and instead of moving data closer to execution, it instead uses thread migration to where the data is. DeX introduces new Linux system calls for execution migration. It captures the thread context including its registers, state, stack, etc., and moves them from one node to another as intended by the system call. The threads can be brought back when they are done executing. To handle state dependent system calls it allows the migrated thread to offload the system call to the original node, requiring no changes to the implementation of other system calls. Along with execution migration, DeX also supports data sharing between nodes.

e: MISC

Leap [46] is a somewhat special proposal. It does not build a memory disaggregation system from scratch, but rather proposes a new page prefetching system of higher accuracy, to be integrated in existing memory disaggregation proposals. To showcase the prefetcher, Leap builds two separate systems, one based on Infiniswap [31], and the other based on Remote Regions [44]. More details about its prefetching techniques are discussed in Section IV-B.

Hydra [47] is equally special as it also does not target innovation in memory disaggregation itself. Instead, it builds upon Remote Regions [44] and Infiniswap [31] to showcase its fault tolerance and compare against existing techniques. More details are discussed in Section IV-C.

2) HYPERVISOR-BASED SYSTEMS

In this subsection we cover proposals that are based on hypervisor implementations or modifications. Since there are only a handful such proposals, we do not further categorize them.

Unlike many other proposals, vNUMA [48] does not make modifications onto existing components. Instead, it is built from scratch with the purpose of running a single virtual machine atop multiple physical nodes. vNUMA is a split architecture Type 1 hypervisor, meaning it is a bare metal hypervisor that does NOT run on top of an operating system. Because of its end goal, vNUMA naturally supports data sharing between different nodes. To this end, it uses a custom network protocol built over Ethernet to move data, using a 4KB page granularity.

Another hypervisor-based implementation is vDSM [49], which is highly similar to vNUMA [48]. vDSM utilizes hypervisor-based page swapping to handle memory disaggregation. Like vNUMA, it also supports data sharing between different nodes in a split mode, but it uses RDMA for its data transfers. A third very similar proposal is GiantVM [50], [51]. GiantVM is a Type 2 hypervisor, running on

top of an operating system. It has an architecture similar to that of vNUMA, with the addition of remote inter-processor interrupt handling, as well as virtual IOs. GiantVM uses RDMA for its data transfers, and was tested with transfers of 4KB and 4MB sizes.

Semeru [52] builds on the same premise of vNUMA [48], although specialized for Java applications. Semeru introduces a pool based system. A compute node runs a Java Virtual Machine (JVM) with limited memory, but allows Java applications running on it to use a distributed universal heap backed by multiple remote memory servers. Semeru also introduces a distributed garbage collector that offloads object tracing to light JVMs running on the memory servers. To handle data swapping between the client and server nodes, Semeru utilizes an existing swapping system called NVMe-oF [53]. Semeru uses 4KB data transfers over RDMA. Further improvements over Semeru were later introduced in MemLiner [54]. MemLiner tries to address the increasing misses and ineffective prefetching resulting from the competition between the application and garbage collector threads. It builds on the key observations that accesses of the applications and the garbage collector are still related (though not temporally), and that the order of which the garbage collector does its tracing is not important. Based on these observations, MemLiner tries to align the working set of the application and garbage collector to significantly reduce remote memory traffic. A third JVM-based garbage collector is Mako [55] which has different design goals from MemLiner and Semeru. While Semeru and MemLiner are both throughput focused, Mako is latency oriented, to better suit the latency sensitive applications of today's datacenters. Similar to Semeru, it also offloads tracing to the memory servers where it executes concurrently with the client parts.

3) SOFTWARE-BASED SYSTEMS

In this subsection, we cover proposals that are mainly software based. We further categorize them based on the style of their software implementations:

a: EXPLICIT SOFTWARE LIBRARY

DLM [56] is a pool-based memory disaggregation system that is fully software based. It uses a user space API for clients and servers. A host file is used to select the memory clients and servers for each deployment. The API is used to allocate and free remote memory, and a signal handler is used to handle segmentation faults (i.e., page faults) and implement a swap system similar to Linux's, albeit completely in user space. Because it is fully in user space, the memory transfer unit, which is the "page size" is also user defined. DLM simply uses TCP for its memory transfers. It also does not support data sharing.

Similar to DLM [56], FaRM [57], GAM [58], and DASC [59] also use the same software library approach, albeit on a split system. One main difference is that they are all based on RDMA. GAM uses one-sided RDMA operations for data, but resorts to two-sided ones for its control flow. Unlike DLM, FaRM and GAM also support data sharing between different compute nodes. FaRM uses data transfers of varying allocation sizes, from 64 bytes to 1MB, while GAM uses 512 bytes, and DASC uses pages of 4KB. Grappa [60], [61], Argo [62], MAGI [63], and rMap [64] also follow somewhat similar approaches. Only the first three support data sharing. Grappa, MAGI, and rMap are RDMA-based, where only MAGI uses two-sided RDMA verbs, while Argo and a second implementation of rMap are MPI based, and thus use TCP as their backend. Grappa keeps user data structures (in reality, user provided malloc sizes) as the unit of data transfers, while MAGI and Argo keep data transfer granularity of 4KB pages, and rMap uses data transfers of multiples of 4KB.

Another approach is that of CODS [65]. CODS is a split system reliant on a PCIe switch to connect multiple nodes. Initially, the nodes negotiate what parts of memory are remote and set up a translation table between their address spaces. CODS uses a Linux driver rather than a real explicit software library. Like DLM [56] the driver offers an API to allocate and free remote memory as needed. Apart from these API calls, memory accesses proceed like normal with no extra user or OS involvement, though they miss in the entire memory hierarchy until they reach the PCIe port.

b: IMPLICIT SOFTWARE LIBRARY

AIFM [66] is somewhat similar to Grappa [60], [61], although on a higher abstraction level. AIFM provides developers with predesigned, C++standard container-like data structures that are *remotable*. These data structures are designed to operate the same way on local memory or remote memory, hidden from the programmer. The AIFM runtime decides when these data structures are kept locally or transferred to a remote server depending on the status of the system. It also encodes information about these data structures and their availability into their pointers. A programmer can force a data structure to exist locally by dereferencing its remotable pointer. AIFM also provides a lower abstraction layer where new data structures can be designed, and also provides the flexibility to run active parts of the data structures on the remote memory, similar to in-memory processing ideas. AIFM uses DPDK [67] with TCP as its memory transfer backend, and the authors leave data sharing for future research.

c: KEY-VALUE STORE

Clover [68] utilizes a persistent memory-based key value store as its memory servers. It proposes passive (i.e., memory only without any processing) nodes as the memory server, and splits all control paths from the data paths by also introducing metadata servers for such tasks. Clover supports data sharing between compute nodes, and uses one-sided RDMA for data transfers and two-sided RDMA for control transfers. Naturally, data transfer sizes in Clover are not uniform sized and depend on the size of accessed data structures. In a subsequent study [69], the authors try to alleviate the conflicts of RDMA writes by using top-of-rack (TOR) programmable switches to serialize such conflicts, as discussed in Section IV-B.

d: PROGRAMMABLE SWITCHES

Two very different software-based approaches are Concordia [70] and MIND [71], which do not execute their software on processors, but on TOR programmable switches. Concordia and MIND assume an existing memory disaggregation system with client and server nodes, with the client nodes including caches for the remote memory. Both proposals utilize the unique location of the network as a mediary between clients and servers to handle cache coherence requests with shorter latencies. The details of these cache coherence systems are discussed in Section IV-B. MIND extends beyond Concordia in that it also uses the TOR switches to handle thread placement, and modifies the Linux kernel on the CPU blades to forward these requests to the TOR switches over TCP, as well as generate RDMA data transfers on page faults.

4) HARDWARE BASED SYSTEMS

In this subsection, we focus on proposals that mainly introduce new hardware designs or modifications. We categorize them as follows:

a: BLADE-BASED

Blade [72], [73] introduces the use of hardware modifications to achieve memory disaggregation. They introduce a pool system where separate compute and memory blades exist. The authors propose two different ways of implementing such a system. The first way is to use unmodified CPU servers of today, connected to memory blades (i.e., memory servers) through a PCIe backplane. This essentially introduces new hardware in terms of the memory blades and their required management hardware such as memory controllers, optional compression or encryption hardware, as well as protection between different sharers/accessors. However, it keeps the CPU blades unmodified. In this implementation, they use hypervisor swapping to transfer pages with a granularity of 4KB, much like MemX [29] and similar designs. The second way assumes a cache coherent interconnect between the CPU and memory blades, and so requires hardware modification on the CPU side to add a coherence filter and interface with the memory. In this case, the data transfer granularity is cache line sized. This does not require modifications to the hypervisor or the OS, beyond the ability to support NUMA memory. In both implementations, there is no data sharing between different CPU nodes, and the authors also discuss no form of fault tolerance for their system. dReDBox [74], [75], [76] follows suit by envisioning an optically connected blade based system similar to Blade's second implementation. However, the studies are more focused on the optical connections and the networking architectures.

Another proposed blade based system is that discussed in MC-DLA [78]. This proposal is somewhat different from other proposals discussed in this paper as it also incorporates GPUs in the disaggregated system, primarily for machine learning applications. MC-DLA uses NVLINKs arranged in a triple ring topology as the communication backend. The introduced blade based system is evaluated using an in house simulator, and so there is no information provided about its associated runtime, if any, the granularity of data transfers, etc.

DirectCXL [77] is the first (and only) proposal we cover that utilizes the newly introduced CXL protocol. DirectCXL realizes normal CPU servers of today as memory hosts, with passive remote memory blades comprising only of DRAM memories along with a CXL enabled memory controller. The connections between these are PCIe based CXL interfaces. Optionally, CXL switches can be introduced into the mix allowing multiple hosts and memories to be connected, although no sharing of resources is allowed between hosts. DirectCXL enumerates the memories at startup time through PCIe, and uses a kernel module/driver to announce them as memory mapped files for user utilization.

Clio [80] is another blade based pooled system. It uses unmodified CPU servers as the compute nodes, but utilizes active memory blades as memory servers. The memory blades comprise DRAM memories, along with an ASIC that handles all the data accesses, as well as a CPU that handles all control operations. The last part is an optional FPGA fabric that can be used to implement any form of near memory computing. The compute node library provides the ability of allocating, freeing, as well as explicitly reading and writing data over Ethernet. Clio supports data sharing between compute nodes.

b: OS

Another study, LegoOS [81] proposes a new split kernel operating system. While not directly proposing a hardware based system, the OS assumes a blades-based system, and deploys a monitor on each blade to keep track it of its utilization and manage it. It utilizes the main memory on the CPU blades as an extra level of caching, and manages this cache in software through the CPU monitors. For address translations, it assumes TLBs and page tables have been entirely moved to the memory blades, and that all CPU caches are virtually indexed and tagged, thus providing a clear compute and memory separation. LegoOS can use multiple blades of each type for any deployed application, and vice versa. However, it does not support sharing data between different CPU blades, and leaves it to the programmer to use messaging if needed. LegoOS provides three different network stacks, based on two-sided and one-sided RDMA, as well as Ethernet, and maintains a data transfer granularity of 4KB. LegoOS uses the existing Linux system call interface for backward compatibility, and also supports fault tolerance through the use of logging to secondary memories as well as to SSDs.

c: FPGA-BASED

PBerry [82] goes a fundamentally different road. It utilizes system on chip FPGAs that include a CPU and an FPGA side by side with a cache coherence interconnect. It then leverages the coherence of the FPGA to provide the illusion of a huge cache representing a remote memory. It does this by only saving the tags on the local FPGA memory, while keeping the

data on the remote memory. It also introduces a kernel module to control this hardware. PBerry is completely focused on the client architecture. It assumes a pool-like system but does not discuss any requirements about the servers or the data transfers between client and server. However, since it is cache coherent, it uses cache lines as the unit of data transfer, in an attempt to reduce dirty data amplification. The same basic principle of PBerry is utilized again in Kona [83]. The FPGA serves as a directory for the far memory, and the local FPGA memory is used as a cache for it. The local cache is the only major component differentiating between Kona and PBerry. Because the FPGA is connected to the cache coherence infrastructure, it can keep track of dirty data and handle misses to remote memory before they cause page faults. Kona uses user-space software, rather than a kernel module, to handle hardware management, as well as remote memory allocation, etc.

ThymesisFlow [84] follows a similar approach to PBerry [82], but it also provides a server implementation. It builds a remote memory client that is connected to the processor through the IBM OpenCAPI [85] cache coherent interconnect. ThymesisFlow also utilizes FPGAs for the implementation, and uses the Xilinx Aurora [86] protocol for its data transfers. On the server end, a similar hardware interface is also connected to the processor through OpenCAPI. ThymesisFlow utilizes a user space application to also manage allocations, distributions, as well as configurations. ThymesisFlow relies on Linux HotPlug capability [87] to support adding and removing memories to users as needed.

d: OTHER HARDWARE MODIFICATIONS

In soNUMA [88], the authors aim to utilize the qualities of RDMA while avoiding its drawbacks, so they build a cache coherent (as opposed to PCIe connected) network interface similar to RDMA and optimized for smaller transfers. They also provide a device driver for the interface, as well as a user API to use it. While, in theory, a system using soNUMA may be designed to share data, no such capability is reported in the study.

DeACT [89] also utilizes hardware changes, although this time for the purpose of providing a shared remote memory (as a resource, not data sharing) without manipulating or modifying virtual memory systems. The insight behind DeACT is that the tasks of remote address translation (translation from a fake physical address inside compute nodes to a real address in the remote memory node) and remote data access control are separate and need not be implemented together. Towards this purpose, it builds a modified memory controller on CPU nodes, where the controller cache translation data can emit a final remote address in its request, without any form of access control checking. The translation data can be saved in the local DRAM where it is reachable by the controller. Once a final translated address has been requested, a central unit responsible for access checking gets involved, and either allows the access or denies it if it does not have the required permissions. This approach is completely invisible from the OS and the user, and allows the OS to manage a fake continuous NUMA style memory space, with no awareness of the underlying remote memory.

Another approach highly similar to DeACT is that taken by MEMSCALE [90], [91]. Like DeACT, MEMSCALE introduces a modified memory controller that can access the memory of other nodes over the HyperTransport protocol. This allows the memory to appear (to the processor and operating system) as a normal physical memory, though with a higher latency. The memories of all nodes in the system are shared, serving as a split architecture. However, MEMSCALE does not support data sharing and was restricted to testing single applications using memory beyond that of a single node.

DVM [92] is another way of implementing disaggregated memory through hardware. DVM introduces a new Instruction Set Architecture (ISA) called DISA. DISA provides a minimal set of Turing complete instructions, as well as registers, and a complete memory model that we will discuss in Section IV-B. DISA is not implemented in hardware, instead it is emulated on top of x86 processors.

Farview [93] is a proposal specific to database engines. It mixes memory disaggregation with the idea of near memory processing. It does so by utilizing FPGAs as the memory controllers, using the memory itself to store the database object, and dividing the FPGA into multiple dynamic regions. Each application accessing the database is assigned a dynamic region that implements the database operator(s) required. Farview augments the hardware implementation by a software library for applications to establish connections and communicate with the FPGA. Farview uses RDMA for data transfers.

5) NETWORK REQUIREMENTS

Unlike most other studies which aim to build memory disaggregation systems and evaluate them, network support [94] and network requirements [95] try to establish the network requirements necessary to support memory disaggregation in the datacenter. Both studies follow the exact same approach by using part of their local memory as an emulated remote memory through a swap space with injected artificial delay. Both studies do not consider data sharing and thus do not include any coherence traffic in the network requirements. The knobs varied in both studies are the amount of injected delay, as well as the percentage of local memory. The difference between both studies is simply their choice of benchmarks. A third study [96] focuses on the feasibility of memory disaggregation by following a different approach. Essentially, it looks at the issue from the application's side, by establishing the requirements needed for application's operation, then comparing those to the specifications of existing networks.

B. MEMORY HIERARCHY

In this subsection we discuss the memory hierarchy of our studied proposals. We only cover proposals that make

TABLE 2. Address space changes and memory hierarchy of disaggregation proposals.

<table>
<tr><th>Paper</th><th>Address Space</th><th>Consis. Model</th><th>Coherence Protocol</th><th>Caching</th><th>Prefetch</th></tr>
<tr><td>FluidMem [40], [41]</td><td rowspan="5">No Change</td><td></td><td></td><td>Software</td><td>Auto On Consec. Access</td></tr>
<tr><td>LegoOS [81]</td><td></td><td>User</td><td>Local DRAM</td><td></td></tr>
<tr><td>Leap [46]</td><td></td><td></td><td></td><td>Adaptive</td></tr>
<tr><td>DeX [45]</td><td>Strong</td><td>Ivy-like</td><td></td><td></td></tr>
<tr><td>Kona [83]</td><td></td><td></td><td>ccFPGA DRAM</td><td></td></tr>
<tr><td>soNUMA [88]</td><td rowspan="9">PGAS</td><td></td><td></td><td></td><td></td></tr>
<tr><td>DASC [59]</td><td></td><td></td><td></td><td></td></tr>
<tr><td>AIFM [66]</td><td></td><td></td><td></td><td></td></tr>
<tr><td>DVM [92]</td><td>Snapshot</td><td></td><td></td><td></td></tr>
<tr><td>FaRM [57]</td><td>Strong</td><td></td><td></td><td></td></tr>
<tr><td>Grappa [60], [61]</td><td>Weak</td><td>Delegate</td><td></td><td></td></tr>
<tr><td>GAM [58]</td><td>PSO</td><td>MSI</td><td>Software</td><td>Single On Sync</td></tr>
<tr><td>Clio [80]</td><td>Release</td><td>User</td><td></td><td></td></tr>
<tr><td rowspan="2">Hotpot [43]</td><td rowspan="3">PGAS like</td><td>Weak</td><td>MRMW</td><td rowspan="3">Software</td><td></td></tr>
<tr><td rowspan="2">Release</td><td>Ivy-like</td><td></td></tr>
<tr><td>Remote Regions [44]</td><td>User</td><td>Single On Command</td></tr>
<tr><td>vNUMA [48]</td><td rowspan="10">Unified</td><td>TSO</td><td>Ivy + Wrtie Propagate</td><td></td><td></td></tr>
<tr><td>GiantVM [50], [51]</td><td rowspan="3">Strong</td><td rowspan="3">Ivy</td><td></td><td></td></tr>
<tr><td>vDSM [49]</td><td></td><td></td></tr>
<tr><td>MAGI [63]</td><td></td><td></td></tr>
<tr><td>Semeru [52]</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Mako [55]</td><td></td><td></td><td></td><td></td></tr>
<tr><td>MemLiner [54]</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Argo [62]</td><td>Weak</td><td>Directory + Self-Invalidate Self-Downgrade</td><td>Software</td><td>Sequence On Miss</td></tr>
<tr><td>Concordia [70]</td><td>Strong</td><td rowspan="2">MSI</td><td></td><td></td></tr>
<tr><td>MIND [71]</td><td>TSO</td><td></td><td></td></tr>
<tr><td>Clover [68]</td><td>KV Store</td><td>Strong</td><td></td><td>KV Meta</td><td></td></tr>
</table>

changes to the address space, or those that introduce data sharing, thus also introducing coherence and consistency to the mix. Table 2 summarizes the comparisons of this subsection.

1) ADDRESS SPACE

a: NO CHANGE

Swapping based systems do not introduce any changes to the addressing or to virtual memory in general. However, for many other proposals, it is natural to either use one unified address space across multiple nodes, or to supply two separate address spaces, one for local and another for remote memory.

b: PGAS

DASC [59], soNUMA [88], and AIFM [66] are all software-based solutions (at least in part, such as for the case of DASC) and thus provide a split address space. The first part of the address space is local and accessible through normal pointers and memory accesses, while the second is remote and usable through API calls. Since all three proposals do not support data sharing, this is a special case of the PGAS addressing discussed in Section II, where the remote part of the address space is not global. The address separation is not as clear in the case of AIFM, as it cannot be defined in a specific range, but is still adequately split to be considered a partitioned address space.

Software based proposals that support data sharing like FaRM [57], Grappa [60], and GAM [58], as well as Clio [80] which uses a hybrid software/hardware implementation, follow the same approach by using a partitioned address space where the remote part is accessible similarly through API calls. However, unlike soNUMA [88], DASC [59], and AIFM [66], they fully conform to the PGAS addressing model, and the remote part of these address spaces are shared between different nodes. DVM [92] is another proposal that uses PGAS addressing, but in a completely different methodology. The instruction set architecture of DVM defines its memory space as a PGAS address space where the remote part is shared across multiple processing units.

c: PGAS-LIKE

Hotpot [43] and Remote Regions [44] are somewhat different. Since their architecture is based on file like-regions, users can access remote memory by mapping these regions or files to memory and using the resulting pointers. In this case, there is no difference between local and remote pointers, and no extra API calls are needed for accessing remote memory. We take the liberty of categorizing these as PGAS-like addressing modes, since there is clear separation in the address space between local and remote spaces. However, unlike usual PGAS-style addressing, this is mostly hidden from the user and does not require any special programming beyond simply opening a remote region or file.

d: UNIFIED

GiantVM [50], vNUMA [48], vDSM [49], MAGI [63], Argo [62], Semeru [52], Mako [55], and MemLiner [54] are also software based implementations, whether in user space or hypervisor. These proposals all allow a single virtual machine or application to span multiple nodes, and thus they all provide a unified address space. The address translation in this case is purely done in software. This is not an issue in the case of hypervisor-based implementations like vNUMA, vDSM, and GiantVM, but can be more work in the case of user space applications. MAGI adds a kernel module for the purpose of handling TLB shootdowns and batches multiple TLB shootdown requests to maintain coherence between different compute node TLBs.

Concordia [70] and MIND [71] also use a single unified address space. However, the translation process happens entirely in programmable switches in which memory is a scarce resource. This forces both proposals to look for a technique that minimizes memory usage. Both proposals split the virtual address space uniformly over physical memory servers, allowing the switch to use one entry per remote memory node for translation.

e: KV

A different case from all of the above is Clover [68] which offers a key-value addressing mode rather than traditional memory addressing, owing to its use of a key-value store as its memory server.

2) CACHING AND PREFETCHING

Caching and prefetching are improvements to existing memory hierarchies, and are not yet popular in memory disaggregation proposals. Only a handful of the proposals presented here support one or the other.

a: CACHING

FluidMem [40] and Argo [62] both provide a software managed cache with page-sized entries. The cache in Argo is restricted to a direct mapped architecture. GAM [58] uses a similar approach by splitting the local memory of its compute nodes to three different parts. The first is left as a local memory for the OS and the application, the second is advertised as a shared global memory, while the third serves as a software managed cache as well as a distributed directory. LegoOS [81] also uses a software monitor on each compute blade to organize and utilize its local memory as a cache. The implementation of the cache's replacement policy and eviction is all built into the node monitor. Hotpot [43] and Remote Regions [44] also provide a software managed cache, although not many details about them are provided.

Clover [68] uses an interesting type of caching. Unlike other proposals, it does not cache the data it may need to access, but rather caches metadata and headers about its hot key-value entries. The cache is not kept coherent, instead, the entries are invalidated using timeouts and epoch-based garbage collection. This cache plays an important role in the consistency model of Clover, as discussed in Section IV-B3.

b: PREFETCHING

FluidMem [40] uses a simple prefetcher that detects consecutive page accesses and conservatively prefetches one page at a time. Argo [62] uses another simple form of prefetching where a sequence of pages are brought in on a single miss. GAM [58] also provides a limited form of prefetching where cache lines are only prefetched on a synchronization operation. Remote Regions [44] leaves it to the programmer to use special API calls to do prefetching when needed, to avoid the latency of page faults.

Leap [46] is a proposal that is fully focused on page prefetching in the context of remote memory systems. It introduces a prefetcher that looks at access history with an adaptive window size, and uses a majority vote to detect past trends. The window size decreases when trends are prevalent, reducing the work needed for next prefetching iterations, and increases on failure to detect a trend, thus using more history data to help detect access trends. It uses a majority vote algorithm called Boyer-Moore to detect the access trends. Once a prefetching decision has been made, it prefetches pages into local buffers, separate from the usual kernel page cache, and eagerly removes pages from this prefetch queue once they have been used, to make room for more prefetches.

3) CONSISTENCY AND COHERENCE

The memory disaggregation proposals we discuss use many different consistency models and coherence protocols. In the following, we categorize these models and protocols and describe them in detail if necessary.

a: STRONG CONSISTENCY

FaRM [57] maintains sequential consistency by assigning IDs to all RDMA transactions, and serializes these transactions to maintain consistency. vDSM [49], MAGI [63], and GiantVM [50] also provide a strong consistency model by using the Ivy memory coherence protocol, discussed previously in Section II. DeX [45] follows a similar approach by providing sequential consistency based on an unspecified Ivy-like MRSW coherence protocol.

While not a memory consistency model in the traditional sense, Clover [68] provides a sequential consistency model for its key-value store. It keeps a linked list of updates for each data structure, forming an update chain. Compute nodes then cache the head of this chain and on every read performs pointer chasing to find the last entry of the linked list. Clover uses skip lists as a shortcut to decrease the amount of pointer chasing needed, but does not eliminate it completely. On writes, the writer appends to the linked list by adding a new entry. It can retire older versions to reclaim space. Clover can suffer from compare and swap like conflicts, where multiples of these requests would be initialized to the same address, and only one would succeed invalidating the others. The authors proposed using programmable switches to cache such requests and maintain their link head address. When a conflict is detected it is resolved by delaying the offending requests, waiting on the first to resolve, then modifying the offending requests with the updated chain head address. This essentially converts a failed request to a successful one without the delay of retrying.

b: TSO/PSO CONSISTENCY

In vNUMA [48], the Ivy protocol is used at its core to keep its memories coherent, but vNUMA improves on it by allowing write propagation for sparse data accesses. Write propagate directly sends writes to the owner node, rather than invalidating their copy and asking for an exclusive copy, as explained in Section II. This results in the downgrade from Ivy's typical sequential consistency to a TSO-like consistency model. GAM [58] also supports caching and uses a PSO memory consistency model by implementing a worker thread based buffering for write requests and allowing reordering, GAM uses an MSI directory based coherence protocol for its caches.

c: WEAK/RELEASE CONSISTENCY

DVM [92] uses a snapshot based consistency model. In essence, all workers would have a consistent view of memory at a specific time (i.e., snapshot). This model allows each sharer to read and write to its own copy of the snapshot without conflicts or disruptions from others. DVM detects these write conflicts, and serializes these conflicts by blocking or retrying writers.

Grappa [60] maintains coherence by keeping ownership of memory pages to nodes. To access any page, requests must

be delegated to its owner node. Because this can potentially put strain on the network as well as unnecessary overhead, Grappa also resorts to batching multiples of these requests together. This batching and delegation results in a weak memory consistency model, but guarantees sequential consistency on synchronization operations. Argo [62] uses a very different approach for memory coherence. It uses a normal directory based cache coherence protocol. However, to ease the coherence traffic pressure off the network, it introduces the idea of self invalidations and self downgrades. Simply put, a cache can decide to self downgrade an exclusive line if it no longer needs it for writing, allowing any subsequent sharing requests from other caches to go through without the downgrade delay as well as its traffic. A similar idea is used to self invalidate cache lines that are no longer being used, so write requests for them can proceed with less network traffic and delay. Argo also uses a weak memory consistency model, and provides synchronization operations to override it if necessary.

Clio [80] uses two different memory consistency models, depending on the API semantics used by the user. The first model is sequential, where all data accesses are blocking and strongly ordered. The second is release consistency based, and uses asynchronous accesses as well as locking and releasing API calls. Clio does not provide coherence by default to avoid its effect on network, and instead leaves it up to the user to handle, the mechanism of which is not discussed. Hotpot [43] provides two different consistency and coherence models as well. To keep its caches coherent, Hotpot introduces two different coherence protocols. The first is a MRMW protocol that uses a weak consistency model and resolves commit conflicts by forcing writers to retry, similar to DVM's [92] snapshot conflict resolution. The second method is an unspecified MRSW protocol, which appears to be similar to the Ivy protocol, but using a release consistency model. Remote Regions [44] chooses to not keep its caches coherent, and instead delegates that task to the programmer. The user can utilize special API calls to either completely clear the cache or completely write back its contents to enforce coherence. These API calls apply to the cache as a whole, not cache lines, and so can be expensive. This approach also results in a release consistency like model.

d: SWITCH BASED

Concordia [70] and MIND [71] both use a simple MSI cache coherence protocol. Concordia also assumes a sequential consistency model, although it can be applied with weaker models to relieve pressure on the switches, while MIND is restricted to a TSO memory consistency model because of the way page faults are handled. Concordia and MIND use programmable switches to act as a directory, using its unique location between clients and servers. Whenever the switch receives a request, it can check the owner node, forward the request to it, or send a multicast signal to multiple sharers, if any. Because of the memory limitations of programmable switches, Concordia only does so for hot cache lines, and leaves the rest to servers. It also provides a migration mechanism by which cache line management/ownership can move between the switch and the servers. In this regard, Concordia can be though of as a cache for the directory itself. MIND handles the switch memory limitations differently, it decouples the data access granularity (fine grained) from the cache coherence granularity (coarse grained), essentially allowing less data to be saved in the directory.

TABLE 3. Fault tolerance models in memory disaggregation proposals.

Paper	Fault Tolerance			
	Replication	Logging	Persistence	Erasure Coding
FaRM [57]	✓	✓		
Infiniswap [31]			✓	
FluidMem [40], [41]	✓			
Hotpot [43]	✓		✓	
GAM [58]	✓		✓	
LegoOS [81]	✓	✓		
FastSwap [32], [33]	✓			
Clover [68]	✓			
XMemPod [34]	✓			
Valet [37]	✓		✓	
Kona [83]	✓			
Hydra [47]				✓

C. FAULT TOLERANCE

In this subsection we discuss the fault tolerance methods used by memory disaggregation proposals. A small subset of our studied proposals actually provide fault tolerance. Most of these opt for the option of replication, although other types of fault tolerance are also used. A summary of these are shown in Table 3.

FaRM [57] uses replicated logging to achieve fault tolerance. It can append these logs to two locations, which can either be in memory or storage form. However, FaRM reports extra overheads when the logging is SSD based. Infiniswap [31] uses remote memory to act as a cache to swap storage devices, which means Infiniswap naturally supports fault tolerance through persistence, while also avoiding its high cost by hiding the delay through caching in remote memory. FluidMem [40] utilizes RAMCloud [42] as its backend, and thus utilizes RAMCloud's existing replication functionality. Hotpot [43] also uses replication for fault tolerance, the degree of which is user defined. Hotpot also innovates in this regard by utilizing existing cached copies of data as replicas for fault tolerance. Hotpot is also based on non-volatile memories, and thus provides persistence naturally. Because its pointers are themselves persistent and guaranteed to be the same across reruns, this also facilitates recovering from faults. GAM [58] also uses logging for fault tolerance. It logs both write accesses as well as coherence accesses as necessary. GAM uses NVRAM and SSD to make its logs persistent. LegoOS [81] does fault tolerance using a mixture of logging as well as replication. It essentially logs accesses to data as well as replicates metadata on a secondary node. Off the critical path, the logs are flushed from memory to storage where it is appended. FastSwap [32], Clover [68], XMemPod [34], and Kona [83] take the easy approach and simply provide replication for fault tolerance.

Valet [37] allows the user to either enable replication to memory, or asynchronous writing to persistent storage to avoid the latency, or both.

Hydra [47] is the only proposal in our list that uses erasure coding. Hydra introduces a resilience manager that is responsible for encoding (or decoding) data, as well as splitting it into the necessary partitions. It then sends these partitions out to different memory devices. Hydra compares its erasure coding implementation to replication-based proposals and shows considerable speedups.

D. DISCUSSION STUDIES

Another class of studies does not introduce implementations, but focuses on discussing the challenges and the future of disaggregated memory. We present these studies here but do not include them in any of our comparison tables.

One such study is [97]. It argues for a software-based implementation and discusses its challenges from that point of view, covering issues like fault tolerance of memory disaggregation systems, memory allocation and placement, memory sharing and memory ordering models, network congestion, and virtual memory delays. Another study is fate sharing [98], which discusses fault tolerance under disaggregated datacenters, and establishes different fate sharing models and how to handle tolerance in each fate sharing model. In [99], the authors also explore the diversity of the design space of memory disaggregation. They talk about multiple design aspects including its opacity from the applications, its dependence on hardware versus software, fault tolerance, as well as data sharing and transfers.

In [100], the authors discuss the transition towards a memory-centric operating systems. They discuss issues and challenges like the use of non-volatile memories, memory sharing, addressing and protection, coherence, and reliability, but do not provide a design or implementation.

E. SUMMARY

We summarize our classification of memory disaggregation studies by quantifying these studies and their umbrella categories. This quantification is shown in Figure 3. Around half of these studies (Precisely 24 out of 50 proposals) are completely implemented in OS or hypervisor, the majority of which fall under the two related categories of swap engines and page fault handlers. The remaining half of the covered studies are a mixture of hardware solutions, software libraries, or both, with a clear bias towards software. We also show that the majority of these solutions do not support data sharing between different client nodes. This is either a constraint forced by the nature of the implementation method used (e.g., swapping cannot be used to share data), or a willful design choice intended to simplify the proposed solution. Lastly, we show that fault tolerance is, for the most part, ignored in our studies. Even when fault tolerance is considered, the simple, and sometimes wasteful, data replication technique is the most popular solution, occasionally mixed with other solutions as well. Precisely one study, Hydra [47], focuses solely on fault tolerance in memory disaggregation systems and introduces an improved technique.

V. OBSERVATIONS AND RECOMMENDATIONS FOR FUTURE WORK

In this section, we present our observations from the covered memory disaggregation studies. We discuss these observations and try to pinpoint areas for improvement and possible future research directions.

It is obvious from the studies we covered that there is a higher preference towards software based implementation, either as user-space libraries or as modules or modifications to operating systems and hypervisors. This is understandable given the relative ease of software development and the possible hardships associated with designing hardware-based solutions. It is also quite common for many of these proposals, hardware and software alike, to NOT support data sharing. This is probably due to two reasons. First, in many of these proposals, especially older ones, remote memory is considered as a faster replacement to storage. Second, allowing data sharing means a higher barrier as it requires significantly more work to develop a functioning memory disaggregation system. A third observation is that for those proposals that support data sharing, there is a higher likelihood of using a relaxed or weak memory consistency models. The reasoning for this is simple, since in many cases memory disaggregation is throttled by the network, trying to alleviate some of that pressure by using weaker memory consistency is only natural.

Diving deeper, in most of the studies we covered, there are two surprising traits that are prevalent. First, there is a tendency in many of these studies to build systems based primarily on intuition and predisposed personal opinion. Intuition is required in some cases, for example, deciding whether a software or hardware approach is best is not an easily quantifiable task, and may require a great deal of intuition. But in many other instances, especially where quantifiable results are possible, intuition should not be the tool of choice. For example, none of the studies we covered show any reasoning for implementing disaggregation in pool or split format. The second observed trait is that many of these studies try to create complete solutions for memory disaggregation. Memory disaggregation is a complex problem with many aspects to consider, and thus we believe it is only natural to break down the task into smaller steps, and approach them individually and incrementally, allowing more care for studying these parts as well as implementing the best solutions for them.

We further recommend the following as directions for future research, based on the two observed traits discussed above as well as other observations discussed within:

- **General System Architecture**: As discussed above, almost none of the reported studies provide any reasoning for choosing a split architecture over a pool-based one, or vice versa. In fact, most studies outright pick one, and completely ignore the existence of the other.

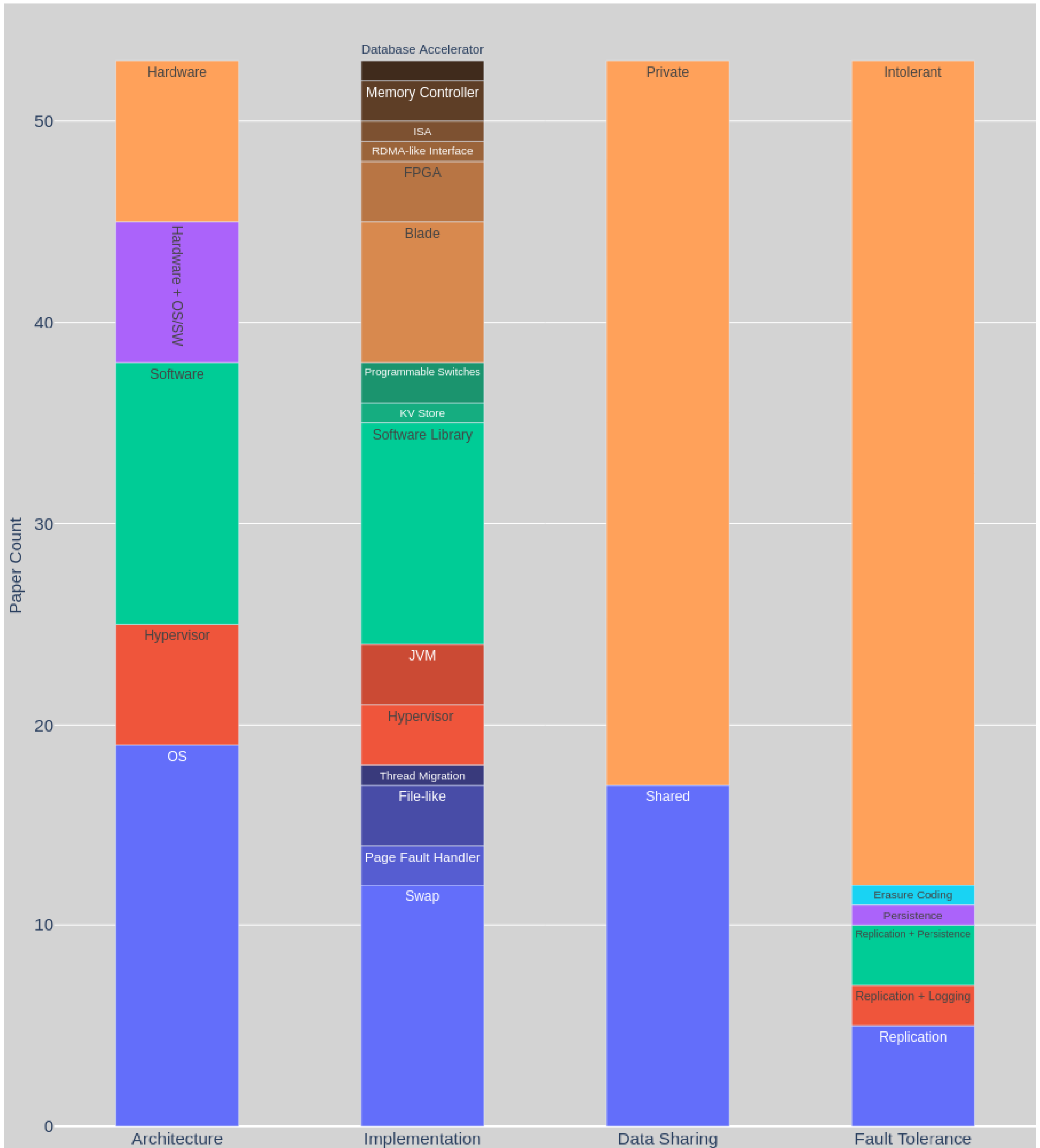

FIGURE 3. Quantifying the papers reviewed in this study and their spanned categories. From left to write, high level architecture, method of implementation, whether data sharing is supported, and the type of fault tolerance supported.

Only one study, [97], acknowledges the existence of both, but quickly dismisses one and focuses on the other. Even disaggregation proposals that compare against other disaggregation proposals mostly compare against previous systems of the same type (e.g., swapping vs swapping based) and thus never provide insight into the high level architecture tradeoffs of different approaches. There is no methodology of research in the field that enables quantifiable comparison between different architectural approaches. We believe this is an important observation that points to the need for building tools (e.g., simulators and common benchmarks) to

enable a dedicated study, and a practical comparison that shows the pros and cons of different approaches.

- **Scope of Disaggregation**: Similar to the high-level architecture of disaggregation systems, this is also a somewhat forgotten aspect in many of the studies. What should be the level on which to do disaggregation in the datacenter? Should it be constrained within the boundaries of a rack? Or should it extend beyond a rack or even groups of racks to cover an entire warehouse scale datacenter? No studies present answers for these questions. While many of the studies we discussed include scale of implementation in their evaluation, it is primarily intended for measuring the scalability of their designs, rather than focus on providing a generalized answer for these questions. Naturally, some studies might come with somewhat restrictive answers based on the technologies they use. We argue that the scope of disaggregation should be studied independently and orthogonally from specific technologies to provide a clear understanding of the benefits gained at every level or scope of disaggregation.
- **Data Sharing**: Today's datacenters run applications inside virtual machines or containers. In many cases, multiple instances of each application are deployed at once, often with some of form of network communication or message passing between them. However, when data sharing is enabled in our proposals, except for those that do not provide coherence, it essentially allows applications to take the form of multi-threading instead. In essence, we argue that supporting data sharing should be studied in that multi-threading context, and should be compared against the usual deployments of today's systems instead of simply comparing to single large memory node deployments. The overheads of data sharing, movement, and coherence should be compared to the overheads of deployment through virtualization, as well as message passing.
- **Memory Hierarchy**: When memory accesses take a long time, the conventional wisdom is to place one or more levels of caching in between the memory and its accessor, obviously aiming at hiding some of this latency. However, the traits of these caching systems realized in our studies seem to be chosen arbitrarily. Studies that implement caching never go beyond one level of caches, which is usually implemented in the whole or part of the local DRAM in a CPU server. The reasoning for only applying one level is never discussed, and it is never established how much caching actually improves the performance and whether its cost is justifiable in terms of latency overheads and complexity. Furthermore, some aspects of the caching hierarchy, such as the choice of consistency models, have clear foundations. Another example is cache line sizes, where it is sometimes justified by using virtual memory or being connected to existing coherence infrastructure. However, some other traits like the cache replacement, the coherence protocols, associativity, etc., are all chosen arbitrarily. We recommend studying the effect of caching along with the effect of all these different factors, to establish the requirements for caching in memory disaggregation systems.
- **Network Requirements for Disaggregation**: The three studies that focus specifically on finding the network requirements for disaggregation all assume a similar split architecture, and base their studies upon the assumption of no data sharing as well as coherence. This may be caused, at least in part, by the relatively early time at which they were done, but it is time to expand upon these studies and find the effect of other types of systems and hierarchies and the strain they put on network.
- **Hardware vs Software**: Many of the proposed solutions we covered are implemented in software, where the argument behind this is that software is easier to adopt, while hardware is expensive to substitute or change. However, very often these solutions require major modifications, if not complete rewrites of applications, which in reality hinders the ease of adopting these software solutions. OS- or Hypervisor-based implementations can avoid changing both hardware and software, although usually at the expense of increased latency caused by page faults and page fault handling. Hardware-based implementations skip all of the unnecessary latencies and can become the most performant of implementations, at the obvious expense of the cost of installing new hardware in the datacenter. That said, datacenter operators already upgrade their hardware every 3 to 5 years [101], making that cost a price paid regardless of disaggregation anyway.
  While not yet widely used, CXL is making its way into more and more datacenter components. Because it is PCIe-based, it is mostly expected to be limited in scale to within a single rack, at most. However, with its components (almost) readily available, we believe that makes it the perfect candidate for future work targeting OS-based or hypervisor-based implementations, allowing a short term commercial realization of memory disaggregation in datacenters. In addition, the birth of promising datacenter suited network stacks like RIFL [102] also create the perfect opportunity to build more performant memory disaggregation systems. Since RIFL has no viable commercial implementations yet, there is enough time for new hardware systems to be researched and studied utilizing RIFL or the like, towards a long term solution that can later be applied in real datacenters.

VI. CONCLUSION

Memory disaggregation is becoming increasingly important as a promising solution to underutilization of datacenter resources. However, by studying the current proposals of memory disaggregation, we discover that a general analysis and quantization of the feasibility and performance of different approaches is severely lacking, with many of the

solutions proposed for memory disaggregation being mostly based on intuition with no real reasoning or explanation offered to their design choices. Furthermore, many such studies are often limited to comparing against prior proposals of the same general approach, offering no insight as to the suitability of any approach over another.

Based on our survey, we believe that future research in memory disaggregation should first start with building the tools necessary (e.g., simulators) to properly study and compare memory disaggregation systems quantitatively and analytically. Memory disaggregation is a complex problem with many interacting components. This in turn makes *fully* decoupling and studying its components not viable. However, we still recommend a divide and conquer approach to studying memory disaggregation whenever possible. Since many of the current proposals try to come up with a complete solution or an entire implementation, these solutions end up either being incomplete and not adequately addressing some aspects of the problem, as well as being inefficient.

REFERENCES

[1] P. Corcoran and A. Andrae, "Emerging trends in electricity consumption for consumer ICT," Nat. Univ. Ireland, Galway, Ireland, Tech. Rep., 2013. [Online]. Available: https://www.researchgate.net/profile/Anders-Andrae/publication/255923829_Emerging_Trends_in_Electricity_Consumption_for_Consumer_ICT/links/00b7d520df6b552e5f000000/Emerging-Trends-in-Electricity-Consumption-for-Consumer-ICT.pdf

[2] N. Su. *The Proportion of Central Energy Consumption in the Global Total Energy Consumption is Increasing Year by Year*. Accessed: Nov. 17, 2022. [Online]. Available: http://tech.idcquan.com/133093.shtml

[3] C. A. Mack, "Fifty years of Moore's law," *IEEE Trans. Semicond. Manuf.*, vol. 24, no. 2, pp. 202–207, May 2011.

[4] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[5] A. M. Potdar, S. Kengond, and M. M. Mulla, "Performance evaluation of Docker container and virtual machine," *Proc. Comput. Sci.*, vol. 171, pp. 1419–1428, Jan. 2020.

[6] H. Liu, "A measurement study of server utilization in public clouds," in *Proc. IEEE 9th Int. Conf. Dependable, Auto. Secure Comput. (DASC)*, Dec. 2011, pp. 435–442.

[7] J. Patel, V. Jindal, I.-L. Yen, F. Bastani, J. Xu, and P. Garraghan, "Workload estimation for improving resource management decisions in the cloud," in *Proc. IEEE 12th Int. Symp. Auto. Decentralized Syst.*, Mar. 2015, pp. 25–32.

[8] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Phoenix, AZ, USA, Jun. 2019, pp. 1–10.

[9] M. Bielski, C. Pinto, D. Raho, and R. Pacalet, "Survey on memory and devices disaggregation solutions for HPC systems," in *Proc. IEEE Int. Conf. Comput. Sci. Eng. (CSE) IEEE Int. Conf. Embedded Ubiquitous Comput. (EUC) 15th Int. Symp. Distrib. Comput. Appl. Bus. Eng. (DCABES)*, Aug. 2016, pp. 197–204.

[10] System Programming Guide. (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Accessed: Nov. 17, 2022. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[11] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proc. 13th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Seattle, WA, USA, Mar. 2008, pp. 26–35.

[12] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[13] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[14] D. L. Weaver, *The SPARC Architecture Manual*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1994.

[15] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 14, no. 2, pp. 434–442, May 1986.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 15–26, May 1990.

[17] A. Potoroaca. (May 12, 2022). *AMD Hits Highest x86 Market Share Ever Amid Desktop CPU Decline*. TechSpot. Accessed: Aug. 17, 2022. [Online]. Available: https://www.techspot.com/news/94565-amd-hits-highest-x86-market-share-ever-amid.html

[18] A. Naeem, A. Jantsch, and Z. Lu, "Architecture support and comparison of three memory consistency models in NoC based systems," in *Proc. 15th Euromicro Conf. Digit. Syst. Design*, Sep. 2012, pp. 304–311.

[19] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synth. Lectures Comput. Archit.*, vol. 6, no. 3, pp. 1–212, Nov. 2011.

[20] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[21] C. Ravishanicar and J. R. Goodman, "Cache implementations for multiple microprocessors," in *Proc. 26th IEEE Comput. Soc. Int. Conf.*, Jan. 1983, pp. 1–5.

[22] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 148–159, May 1990.

[23] D. Sharma and S. Tavallaei, "Compute express link 2.0 white paper," CXL Consortium, 2020.

[24] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 2010.

[25] T. Lahiri, M.-A. Neimat, and S. Folkman, "Oracle TimesTen: An in-memory database for enterprise applications," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, Jun. 2013.

[26] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1401–1412, Sep. 2016.

[27] D. Zahka and A. Gavrilovska, "FAM-graph: Graph analytics on disaggregated memory," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Jun. 2022, pp. 81–92.

[28] OpenAI. (Nov. 30, 2022). *ChatGPT: Optimizing Language Models for Dialogue*. Accessed: Feb. 10, 2023. [Online]. Available: https://openai.com/blog/chatgpt/

[29] M. R. Hines and K. Gopalan, "MemX: Supporting large memory workloads in Xen virtual machines," in *Proc. 2nd Int. Workshop Virtualization Technol. Distrib. Comput. (VTDC)*, Reno, NV, USA, Nov. 2007, pp. 1–8.

[30] A. Samih, R. Wang, C. Maciocco, M. Kharbutli, and Y. Solihin, "Collaborative memories in clusters: Opportunities and challenges," in *Transactions on Computational Science XXII*, vol. 8360, M. L. Gavrilova and C. J. K. Tan, Eds. Berlin, Germany: Springer, 2014, pp. 17–41.

[31] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Symp. Networked Syst. Design Implement.*, 2017, pp. 649–667.

[32] W. Cao and L. Liu, "Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 191–200.

[33] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1664–1673.

[34] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Trans. Comput.*, vol. 69, no. 6, pp. 844–855, Jun. 2020.

[35] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Providence, RI, USA, Apr. 2019, pp. 317–330.

[36] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proc. 15th Eur. Conf. Comput. Syst.*, Heraklion, Greece, Apr. 2020, pp. 1–16.

[37] J. Bae, G. Su, A. Iyengar, Y. Wu, and L. Liu, "Efficient orchestration of host and remote shared memory for memory intensive workloads," in *Proc. Int. Symp. Memory Syst.*, Washington, DC, USA, Sep. 2020, pp. 194–208.

[38] S. Cui, L. Jin, K. Nguyen, and C. Wang, "SemSwap: Semantics-aware swapping in memory disaggregated datacenters," in *Proc. 13th ACM SIGOPS Asia–Pacific Workshop Syst.*, Singapore, Aug. 2022, pp. 9–17.

[39] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and adaptive swapping for multi-applications on remote memory," 2022, *arXiv:2203.09615*.

[40] B. Caldwell, Y. Im, S. Ha, R. Han, and E. Keller, "FluidMem: Memory as a service for the datacenter," 2017, *arXiv:1707.07780*.

[41] B. Caldwell, S. Goodarzy, S. Ha, R. Han, E. Keller, E. Rozner, and Y. Im, "FluidMem: Full, flexible, and fast memory disaggregation for the cloud," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Dec. 2020, pp. 665–677.

[42] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, and D. Ongaro, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–55, Aug. 2015.

[43] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proc. Symp. Cloud Comput.*, Santa Clara. CA, USA, Sep. 2017, pp. 323–337.

[44] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, and A. Ramanathan, "Remote regions: A simple abstraction for remote memory," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 775–787.

[45] S.-H. Kim, H.-R. Chuang, R. Lyerly, P. Olivier, C. Min, and B. Ravindran, "DeX: Scaling applications beyond machine boundaries," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Dec. 2020, pp. 864–876.

[46] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 843–857.

[47] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin, "Hydra: Resilient and highly available remote memory," in *Proc. 20th USENIX Conf. File Storage Technol.*, 2022, pp. 181–198.

[48] M. Chapman and G. Heiser, "vNUMA: A virtual shared-memory multiprocessor," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 349–362.

[49] Z. Ding, "VDSM: Distributed shared memory in virtualized environments," in *Proc. IEEE 9th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Nov. 2018, pp. 1112–1115.

[50] J. Zhang, Z. Ding, Y. Chen, X. Jia, B. Yu, Z. Qi, and H. Guan, "GiantVM: A type-II hypervisor implementing many-to-one virtualization," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, Lausanne, Switzerland, Mar. 2020, pp. 30–44.

[51] X. Jia, J. Zhang, B. Yu, X. Qian, Z. Qi, and H. Guan, "GiantVM: A novel distributed hypervisor for resource aggregation with DSM-aware optimizations," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, pp. 1–27, Mar. 2022.

[52] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "*Semeru*: A memory-disaggregated managed runtime," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement.*, 2020, pp. 261–280.

[53] D. Minturn, "NVM express over fabrics," in *Proc. 11th Annu. OpenFabrics Int. OFS Developers' Workshop*, 2015. [Online]. Available: https://downloads.openfabrics.org/downloads/Media/Monterey_2015/Monday/monday_10.pdf

[54] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu, "MemLiner: Lining up tracing and application for a far-memory-friendly runtime," in *Proc. 16th USENIX Symp. Operating Syst. Design Implement.*, 2022, pp. 35–53.

[55] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu, "Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters," in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, San Diego, CA, USA, Jun. 2022, pp. 92–107.

[56] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato, "DLM: A distributed large memory system using remote memory swapping over cluster nodes," in *Proc. IEEE Int. Conf. Cluster Comput.*, Oct. 2008, pp. 268–273.

[57] A. Dragojevi, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement.*, 2014, pp. 401–414.

[58] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang, "Efficient distributed memory management with RDMA and caching," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, Jul. 2018.

[59] C.-S. Li, H. Franke, C. Parris, and V. Chang, "Disaggregated architecture for at scale computing," in *Proc. 2nd Int. Workshop Emerg. Softw. Service Anal.*, Lisbon, Portugal, 2015, pp. 45–52.

[60] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular applications," in *Proc. Int. Workshop Rack-Scale Comput.*, 2014.

[61] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 291–305.

[62] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas, "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, Portland, OR, USA, Jun. 2015, pp. 3–14.

[63] Y. Hong, Y. Zheng, F. Yang, B.-Y. Zang, H.-B. Guan, and H.-B. Chen, "Scaling out NUMA-aware applications with RDMA-based distributed shared memory," *J. Comput. Sci. Technol.*, vol. 34, no. 1, pp. 94–112, Jan. 2019.

[64] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on HPC systems," in *Proc. IEEE 32nd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Sep. 2020, pp. 183–190.

[65] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang, "Cost effective data center servers," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 179–187.

[66] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement.*, 2020, pp. 315–332.

[67] H. Zhu, *Data Plane Development Kit (DPDK): A Software Optimization Guide to the User Space-Based Network Applications*. Boca Raton, FL, USA: CRC Press, 2020.

[68] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 33–48.

[69] S. Grant and A. C. Snoeren, "In-network contention resolution for disaggregated memory," in *Proc. Workshop Resour. Disaggregation Serverless (WORDS)*, 2020.

[70] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed shared memory with in-network cache coherence," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 277–292.

[71] S. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: In-network memory management for disaggregated data centers," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, Oct. 2021, pp. 488–504.

[72] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, 2009.

[73] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, Feb. 2012, pp. 1–12.

[74] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dReDBox project vision," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2016, pp. 690–695.

[75] K. Katrinis, G. Zervas, D. Pnevmatikatos, D. Syrivelis, T. Alexoudi, D. Theodoropoulos, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "On interconnecting and orchestrating components in disaggregated data centers: The dReDBox project vision," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2016, pp. 235–239.

[76] G. Zervas, H. Yuan, A. Saljoghei, Q. Chen, and V. Mishra, "Optically disaggregated data centers with minimal remote memory latency: Technologies, architectures, and resource allocation," *J. Opt. Commun. Netw.*, vol. 10, no. 2, p. A270, Feb. 2018.
[77] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, high-performance memory disaggregation with DirectCXL," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 287–294.
[78] Y. Kwon and M. Rhu, "A disaggregated memory system for deep learning," *IEEE Micro*, vol. 39, no. 5, pp. 82–90, Sep. 2019.
[79] NVIDIA. *NVLink & NVSwitch for Advanced Multi-GPU Communication*. Accessed: Feb. 11, 2023. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/
[80] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, Feb. 2022, pp. 417–433.
[81] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement.*, 2018, pp. 69–87.
[82] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam, "Project PBerry: FPGA acceleration for remote memory," in *Proc. Workshop Hot Topics Operating Syst.*, Bertinoro, Italy, May 2019, pp. 127–135.
[83] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 79–92.
[84] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "ThymesisFlow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 868–880.
[85] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM J. Res. Develop.*, vol. 62, no. 4/5, pp. 8:1–8:8, Sep. 2018.
[86] Xilinx. (2010). *Aurora 8B/10B Protocol Specification*. [Online]. Available: https://docs.xilinx.com/v/u/en-U.S./aurora_8b10b_protocol_spec_sp002
[87] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri, "Linux kernel hotplug CPU support," in *Proc. Linux Symp.*, vol. 2, 2004.
[88] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 3–18, Feb. 2014.
[89] V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "DeACT: Architecture-aware virtual memory support for fabric attached memory systems," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Mar. 2021, pp. 453–466.
[90] H. Montaner, F. Silla, H. Froning, and J. Duato, "MEMSCALE^{TM}: A scalable environment for databases," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Sep. 2011, pp. 339–346.
[91] H. Montaner, F. Silla, and J. Duato, "A practical way to extend shared memory support beyond a motherboard at low cost," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, Chicago, IL, USA, Jun. 2010, pp. 155–166.
[92] Z. Ma, Z. Sheng, and L. Gu, "DVM: A big virtual machine for cloud computing," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2245–2258, Sep. 2014.
[93] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. Milojičić, and G. Alonso, "Farview: Disaggregated memory with operator off-loading for database engines," 2021, *arXiv:2106.07102*.
[94] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proc. 12th ACM Workshop Hot Topics Netw.*, College Park, MD, USA, Nov. 2013, pp. 1–7.
[95] P. X. Gao, A. Narayan, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement.*, Nov. 2016, pp. 249–264.
[96] P. S. Rao and G. Porter, "Is memory disaggregation feasible? A case study with spark SQL," in *Proc. Symp. Architectures Netw. Commun. Syst.*, Santa Clara, CA, USA, Mar. 2016, pp. 75–80.
[97] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proc. Symp. Cloud Comput.*, Santa Clara, CA, USA, Sep. 2017, pp. 121–127.
[98] A. Carbonari and I. Beschasnikh, "Tolerating faults in disaggregated datacenters," in *Proc. 16th ACM Workshop Hot Topics Netw.*, Palo Alto, CA, USA, Nov. 2017, pp. 164–170.
[99] N. Pemberton and U. C. Berkeley. (2019). *Exploring the Disaggregated Memory Interface Design Space*. Accessed: May 18, 2022. [Online]. Available: http://word19.ece.cornell.edu/word19-paper8-camera.pdf
[100] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond processor-centric operating systems," in *Proc. 15th Workshop Hot Topics Oper. Syst. (HotOS XV)*, Kartause Ittingen, Switzerland, 2015, pp. 1–7.
[101] H.-A. Ounifi, X. Liu, A. Gherbi, Y. Lemieux, and W. Li, "Model-based approach to data center design and power usage effectiveness assessment," *Proc. Comput. Sci.*, vol. 141, pp. 143–150, Jan. 2018.
[102] Q. Shen, J. Zheng, and P. Chow, "RIFL: A reliable link layer network protocol for data center communication," *J. Opt. Commun. Netw.*, vol. 14, no. 3, pp. 111–126, Mar. 2022.

MOHAMMAD EWAIS (Graduate Student Member, IEEE) received the bachelor's degree in electrical and computer engineering from Alexandria University, Egypt, and the master's degree in computer engineering from The University of British Columbia, Vancouver, BC, Canada. He is currently pursuing the Ph.D. degree in computer engineering with the University of Toronto, Toronto, ON, Canada. His research interests include computer architecture, reconfigurable computing and FPGAs, and datacenter architecture.

PAUL CHOW (Life Fellow, IEEE) is currently a Professor with The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto. His research interests include reconfigurable computing with an emphasis on programming models, middleware to support programming and portability, and scaling to large-scale, and distributed FPGA deployments. He has been the Technical Program Chair and the General Chair of FPGA, the premier conference on FPGAs, and FCCM, the main conference for reconfigurable computing. He co-founded AcceLight Networks to build a high-capacity, carrier-grade, and optical switching system. He is also the Co-Founder of ArchES Computing Systems, which is developing reconfigurable computing technology for the data center.