Building for Disaggregated (Persistent) Memory

Karan Grover Sambhav Satija Vinay Banakar

744 Final Report – Group 16

Abstract

Datacenters have started supporting newer hardware like PM (Persistent Memory) and fast RDMA (remote direct memory access) using Infiniband. To effectively exploit the capabilities of such hardware, software often needs to be redesigned. In this report we first outline a concrete set of problems which would benefit from a hardware-aware redesign. Second, we present our learnings from building two key RDMA-enabled applications. We evaluate these applications to determine the effect of RDMA on scalability and performance, and PM's effect on fault tolerance.

1 Introduction

Better hardware has always necessitated better software. As hardware improves, the overlaying software's overheads become more apparent. The adoption of SSDs required changes to filesystems[21] and applications[16]. Accelerators like GPUs drove the need for software[18, 25] which can utilize them effectively.

In recent times, we have seen drastic improvements in two domains: persistence and network hardware. Intel is the first company to develop consumer-grade NVMe (Non-Volatile Memory) 3D-XPoint Optane devices[24] (referred to as PM from here onwards), which promise persistence at DRAM latencies. On the networking front, the emergence of fast fabric like Infiniband has finally made RDMA[17] practical. Datacenters have started deploying this new hardware[23], which is tightly coupled with academic efforts to build systems which can fully exploit such hardware[26, 7].

The marriage of fast RDMA and PM yields an interesting design opportunity: disaggregated PM. Disaggregating PM would provide the attractive performance characteristics of PM while improving resource utilization. Improving utilization of new 'fancy' hardware is paramount to reduce the friction of adoption. We find that disaggregated PM is better than the sum of its parts: it can easily enable new functionality such as fault tolerance.

Contributions:

- We present a set of workloads which would have clear gains if implemented using a disaggregated PM architecture. Each of these workloads are potential future projects. We outline them to demonstrate the breadth of problems which would benefit from disaggregated PM **(§3)**.
- We present our journey of deploying RDMA in Cloudlab and implementing our own multi-client framework for disaggregated memory **(§4)**.
- We present our experience of building and designing RDMA-aware applications **(§5)**.
- We evaluate these applications for their scalability and performance **(§6)**.

Deviations from expectations: We originally wished to do our study on real RDMA+PM hardware. However, we had access to Cloudlab which did not have PM devices, but did support Infiniband RDMA. Since PM performance is close to DRAM[27], our implementations use DRAM and are effectively disaggregeted-memory models. The Linux kernel currently supports Optane DC pmem

over RDMA [3], thus we expect our system implementation and evaluation to carry over. This has the additional benefit of us being able to actually learn a new technology like RDMA.

2 Background

2.1 Intel's PM offering

3D-XPoint is Intel's PM consumer-grade solution with reads just 2X slower than DRAM and write latencies matching DRAM[27] while also providing persistence. It further promises higher density, better power efficiency, and lower cost per byte than DRAM, thereby offering great potential for improving the performance of data-intensive software systems. PM makes it possible to work with large datasets in memory; with fast decompression techniques, spinning disk accesses can now be completely avoided. PM is poised to surpass the shipped DRAM capacity in the next six years [1], making it critical to focus on building PM-specific application.

2.2 Infiniband RDMA

Advancements in Inifiniband smart NICs and switches has made RDMA fast and thus, practical. In RDMA, the local CPU is not involved with the data copy and hands the copy operation to its smart NIC. Infiniband supports 2 different kinds of paradigms: a message passing interface and a pure remote-local memory copy interface.

Both the sender and the receiver register their memory regions with their smart NICs. This sets up virtual page tables within the smart NICs which allows them to directly interface with the IMC (integrated memory controller) without involving CPU.

RDMA operations: The *send* operation allows the sender to transmit a message to a remote QP(QueuePair)'s receive queue. The sender does not have any control over where the data will reside on the remote host. Symmetrically, in *recv*, a client reads the message sent by a peer. It controls where this message is copied in its local memory.

In *RDMA read*, the caller specifies the remote virtual address as well as a local memory address to be copied to. *RDMA write* is similar to *RDMA read*, but the data is written to the remote host. *RDMA read/write* operations are performed with no notification to the remote host, thus the remote CPU is kept idle.

In Table 1, we observe that the latency for *RDMA write* operations is slightly(but consistently) less than the *send* operation. This is because a *send* operation requires trapping into the remote CPU while the remote NIC can acknowledge an *RDMA write* directly without involving its CPU.

2.3 The Union: Disaggregated PM

Resource utilization in data center can be very low due to over-provisioning and inflexible allocations [20], and the optimal hardware mix must evolve to meet the changing workloads [10]. Disaggregated hardware provides these exact features. The ability to independently scale any resource based on the varying demands and each resource being accessible across the datacenter (with varying performance due to locality) allows quick adoption and effective utilization of new hardware. PM exposed using RDMA enables a disaggregated PM model which allows infinitely scalable storage with byte addressable persistence at low latency and high throughput.

2.4 External Merge Sort

The Merge-sort algorithm for external memory uses the sort and merge strategy to sort the huge data file on external memory. It sorts chunks that fit in main memory and then merges the sorted chunks into a single larger file. Thus it can be divided into 2 phases – Run formation Phase (Sorting chunks) and Merging Phase. In Run phase, n blocks of data are scanned, one memory load at a time. Each memory load consisting of M blocks is sorted into a single run and is given as output to a series of stripes on the disk. Thus there are n/M runs each sorted in stripes on the disk. After initial runs are formed, the merging phase begins where groups of R or M/B runs are merged. For each merge, the R runs are scanned and merged in an online manner as they stream through the internal memory (as shown in fig 3). So each merging step will take $O(n/B)$ IO and the number of runs decrease by a factor of

Transaction	Min Lat.(us)	Max Lat.(us)	Avg Lat.(us)	99% Lat.(us)	99.9% Lat.(us)
Send	1.65	5.24	1.71	1.77	5.24
RDMA Read	3.01	7.67	3.13	3.35	7.67
RDMA Write	1.61	3.19	1.64	1.69	3.19
Atomic (C&S)	3.05	7.46	3.18	3.39	7.46

Table 1: Latency Statistics for different RDMA Transactions. (*Recv* unsupported by benchmark tool.)

M/B causing the total number of merge levels to be $\log_{M/B} n/M$. Finally, since each level has n/B IOs and $\log_{M/B} n/M$ phases, the total number of IO is,

$$O\left(\frac{n}{B} log_{\frac{M}{B}} \frac{n}{B}\right)$$

3 Workloads

We spent a significant amount of time searching for workloads which would clearly benefit from using disaggregated memory. We outline our ideas here:

1. **Reducing shuffles in serverless data analytics:** There has been past work to enable data processing on serverless frameworks[12]. The infrastructure flexibility provided by serverless frameworks works well with ever-changing data processing demands, thus offering high elasticity and performance at low cost. However, shuffles between nodes are still needed which are typically done over remote blob stores like S3. Blob stores provide an inherently restrictive API where the access granularity is of the order of MBs or more. Using a disaggregated pmem store will allow serverless instances to pass intermediate data by simply passing references. Furthermore, the finer granularity and inherent fault-tolerance provided by PM can allow us to revisit the need of in-memory dataflow frameworks like Spark. Additionally, this provides the flexibility for workers to save their output and terminate, resuming only after other input required for further steps becomes available.

2. **Reducing serverless startup time:** Cloud function startup times are a critical performance measure [19]. To maintain a certain latency target, developers today resort to over provisioning/replicas, which is often wasteful and not helpful [22]. With disaggregated PM pools, pre-initialized memory regions could reside in the shared memory that would enable near instant start times without requiring any replicas.

3. **Better fault recovery using faster checkpoints:** Long running scientific workloads (simulations) face a constant juggling act of determining when to spend time checkpointing state v/s when to defer checkpointing at the cost of slower fault recovery. Disaggregated PM can act as a remote memory heap for such processes, alleviating the need for explicit checkpoints. Any crashes in the simulator process does not affect the heap, or memory state, and the process can be effectively resumed with the same state. Stage based checkpoints can also be enabled by background tasks running on the remote disaggregated store which can provide checkpoints with copy-on-write mechanisms.

4. **Parameter servers in Machine Learning training**: Distribution of parameters in a distributed learning environment is limited by the shuffle performance. This is exacerbated by the ever-increasing size of weights that need to be shuffled, and the redistribution of weights in the face of faulty nodes. Parameter servers obviate the need of distributing weights between workers by acting as the single source of truth for weights. With byte-level granularity, clients can now access particular set of parameters with easy and take advantage of the shared memory paradigm by allowing models to update the parameters in place safely.

```
byte buffer[32*1024]; // pinned communication buffer

// triggers a copy from remote address into buffer.
void bd_recv(void *remoteAddr, size_t len, context *ctx);
WAIT(ctx); // blocks until copy completes


memcpy(buffer, data, sizeof(data)); // ready data for copy.
// triggers a copy from buffer into remote memory.
void bd_send(void *remoteAddr, size_t len, context *ctx);
WAIT(ctx); // blocks until copy completes
```

Figure 1: Our interface mimics `memcpy`

4 Building an RDMA framework

While we have a list of possible systems we can build, we wanted to first learn and systematically evaluate the characteristics of RDMA-enabled systems. To do that, we focused on building and evaluating remote-memory aware systems from the ground up.

We begin by discussing how we set up and tested the RDMA infrastructure. We describe the installation methodology and basic benchmarks in § 4.1, talk about how we created a convenient wrapper for RDMA Reads & Writes in § 4.2 and go through a proof-of-concept Binary Search Tree in § 4.3.

4.1 Infrastructure setup

We have setup our infrastructure on Cloudlab instances with ConnextX-4 adapters which support RDMA (Infiniband). While these machines do not have PM, prior work has shown that PM has similar latency to DRAM. Thus, our remote storage servers expose their local DRAM. This compromise, however, allows us to get the experience of building RDMA-centric systems.

We installed the Mellanox OFED drivers on 3 such Cloudlab machines. We ran a few performance tests to ensure that the infrastructure was working correctly. In our tests we evaluated the latency and bandwidth of the following RDMA operations: Send Transactions, Read Remote Memory, Write Remote Memory and Atomic Transactions. We run scripts provided by Mellanox that stress the system by generating synthetic tests. We present these baseline results in Table 2 and Table 1.

Transaction	BW Peak(MB/s)	BW Avg(MB/s)
Send	1103.39	1103.39
Read	1103.45	1103.45
Write	1103.37	1103.37
Atomic (C&S)	16.99	16.99

Table 2: Peak and Avg Bandwidth of different RDMA Transactions

4.2 RDMA Reads & Writes

To allow applications to read from and write to remote memory, we created a multi-threaded state-driven framework which manages the messy NIC interactions. We exposed our own custom API to the developers (ref. Fig 1). We have designed the programming interface to be similar in style to the APIs used to interact with local memory. We expose `bd_recv`, `bd_send` and `WAIT` calls. `WAIT` is implemented as a spinlock which is released once the NIC acks the transfer. We also have variants of `bd_send` and `bd_recv` which permit the programmer to have finer control of the offset of communcation buffer being used.

As a first step to verify if RDMA is working properly, we use this API to write data to a remote memory location and then read the same location to verify that we get the same data. To do this, we implement two programs - 'server' and 'client'. We show that the client can write an object to the remote server's memory and read the same object back. Moreover, we also verify that our program supports multiple clients reading from the same 'server'. We assume that concurrent clients will not write to the same address. Principally, atomicity in concurrent

modifications is not provided by RDMA+PM and has to be achieved out of channel. This has been historically achieved using either 'RDMA-atomic' based locks or distributed locks.

4.3 Binary-Search Tree

Having implemented the basic primitives of read and write to remote memory, we can build complex data structures. We present a complete TreeSet data-structure, implemented as a binary tree. While simple, this end-to-end implementation demonstrates the completeness of our remote read-write framework. Our prototype allows multiple readers to concurrently access the tree, and writers can update the set.

For our implementation, we designed an extremely simple btree interface which allows us to insert and search nodes in the treeset data-structure.

```
insert_node(100, ctx, 0); // root node
insert_node(25, ctx, 0);
insert_node(125, ctx, 0);
...

foundIdx = find_node_idx(25, ctx, 0);
assert(foundIdx == 1);
foundIdx = find_node_idx(125, ctx, 0);
assert(foundIdx == 2);
...
```

With this, we show that our framework is complete enough to support higher level abstractions.

5 Designing RDMA-aware applications

To validate the effectiveness of using disaggregated-PM for a wide variety of systems, we pick two workloads which while being completely different from each other, are still central to most modern big data systems: indexing and sorting.

5.1 Indexing using B+-Tree

A B+ tree is an n-ary tree where each leaf stores a segment of sorted key/value elements. It supports key lookup queries while variants can support lock free updates as well [14]. Our workload is as follows: a B+tree is accessible to all clients via disaggregated memory and they directly run search queries on remote memory (ref Fig 2).

The key benefit of such an architecture is ease of propagating updates. Had each client maintained a local copy of the index tree, updates would need to be gossiped and ensuring consistent reads across nodes would have been non-trivial. Having a single source of truth cleanly circumvents the problem of stale or inconsistent reads while minimizing the cost of updates.

A client executing a search query would need to traverse the tree which requires pointer chasing. Thus we stumble upon our first problem: ***RDMA-addressable pointers***. Typically, one would store, say 16, virtual addresses to children nodes in the form of `node_t* children[16]`. Such virtual addresses are however only valid for remote CPU accesses and not remote RDMA-accesses.

To solve this we implement a stack allocator (arena allocator is left for future work), and children now store offsets into the allocator region instead of virtual addresses. This enables RDMA access by simple pointer arithmetic. Our `Node` data layout and `BPTree` access API is as follows:

```
typedef uint64_t offsetPtr_t;
// offset into allocated region to enable
    RDMA-addressable virtual memory

class Node {
public:
  bool IS_LEAF;
  int64_t key[16]; // 16 is configurable
  int value[16 * VALUE_SIZE_IN_4BYTES];
  offsetPtr_t childPtrs[16 + 1];
};

class BPTree {
  int search(int64_t);
  void insert(int64_t, int);
};
```

After implementing the B+ Tree, we observed that there was scope for optimizations. We specifically make two optimizations to our implementation:

Caching Top Levels: We observe that we can trade some client memory and save on network cost for each search query. Since B+-tree typically has a large fanout parameter (16, in our case), the depth of

the tree grows slowly even for large datasets. Even for a tree indexing 1 trillion key/values, the depth is 9. Caching even 2 layers of the tree will shave off more than 20% of the cost while only requiring space for 17 nodes, a clear win!

In our implementation, we cache the top 2 levels of the B+ Tree. Accesses to nodes deeper inside the tree result in remote RDMA reads. We can vary the caching depth based on the trade-off between local memory and network cost.

Different caching policies like LRU can greatly improve the performance of skewed reads workloads, though we leave a systematic evaluation for future work. It is important to realize that such an optimization makes sense either for a mostly-static dataset where cached values can be trusted or where loose consistency is tolerated.

Selectively getting required value in leaves: We observe that a) our `Node` data layout reserves space for values even for non-leaf nodes, and b) we pull in all the values for a leaf we are interested in. We restructure the data layout to push the value at the end of the struct:

```
class Node {
  int64_t key[16];
  int value[16 * VALUE_SIZE_IN_4BYTES];
  ...
};
 <!-- transformed into -->
class Node {
  int64_t key[16];
  ...
  int value[16 * VALUE_SIZE_IN_4BYTES];
};
```

This allows us to prevent fetching the values for all nodes by simply fetching a smaller size into memory. For leaf nodes, we do pointer arithmetic to determine the precise location of the required value and surgically extract it from remote memory, thus minimizing any network overheads. We show in §6.1 that this reduces the network transfer by over 90%.

5.2 Distributed Merge Sort

We implemented a recursive binary **Distributed-Merge** sort to illustrate the shared memory communication benefits the disaggregated pool and one-sided RDMA provides. The algorithm has two stages, the `RUN` and `MERGE`, similar to that an External-Merge sort described in 2.4. The unsorted file resides on a secondary storage which gets loaded to memory/PM as the server initiates. let, SM be the size of unsorted records. This memory region is registered with all the clients that establish connection with the server, let this be AM. Along with the file data we also register auxiliary memory equal to that of the file size with all the clients. This allows us to provide crash recovery for clients which will enable them to continue merge on restart.

Each client on the other hand has a limited amount of memory available. If CM is the registered memory of the client with server, we assume that at least $4 * CM$ of memory is available on the client. During the `RUN` phase, all clients copy CM amount of data from SM at different offsets. Each client sorts the local memory and writes it back to AM. The server on receiving a disconnect request

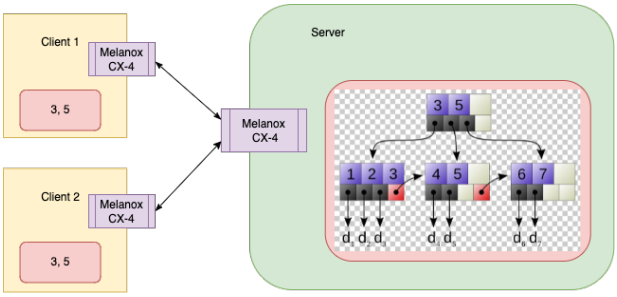

Figure 2: B+Tree remote queries

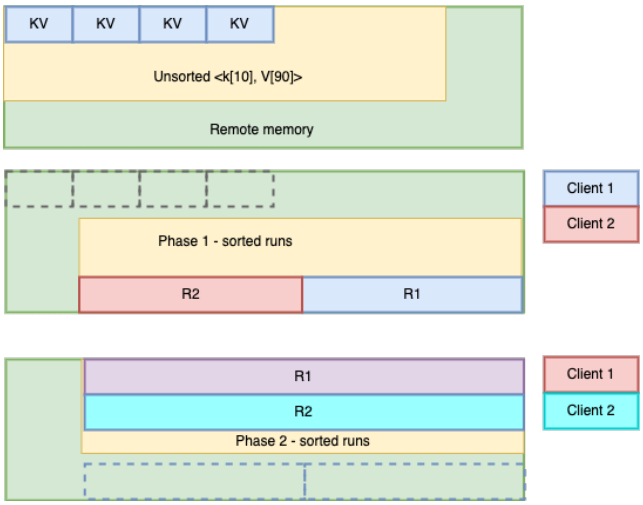

Figure 3: Instant crash recovery during sorting

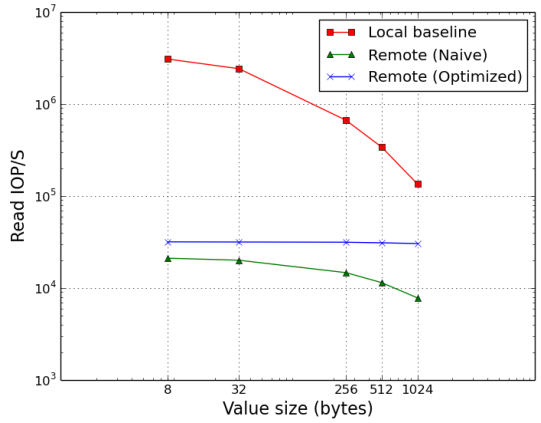

Figure 4: B+Tree (local v/s remote queries)

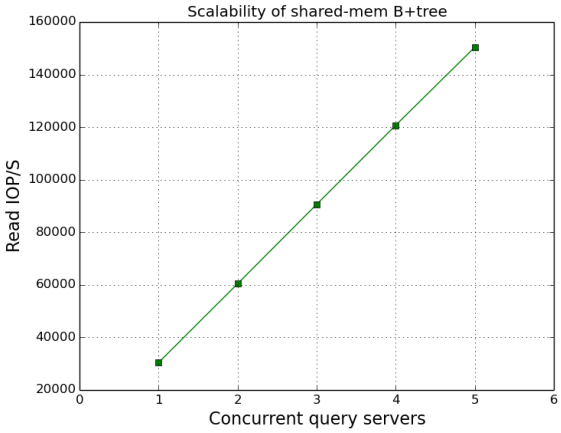

Figure 5: B+Tree (strong scaling w/ increasing concurrency)

from client flushes the sorted set to secondary storage. In total, there are SM/CM number of sorted runs generated, equal to the number of clients that worked on it parallel. On real PM we could have skipped writing these runs to disk. This marks the end of `RUN` stage.

During the `MERGE` stage, each client merges two run files to form a sorted larger run file. We chose not to perform multi-way merging because it can be bottlenecked by the single client throughput. In the favour of parallelism we perform more merges. Hence, our algorithm requires $\log_2(SM/CM)$ number of merge phases. In each phase, the client reads data from two runs in CM chunks and loads it to two buffers. Where the merged output of these buffers are put to an output buffer. Once the output buffer becomes full, it is written back to the remote pool at a new offset as shown in Fig 3. The server waits for the last client to disconnect which is when it flushes all the data to appropriate run files on disk.

The number of merge phases and network requests can be controlled using CM; client's local memory. A larger CM would reduce the number of merges but also decreases parallelism. This trade off if a function of cost of network access, single client throughput and amount of data to be sorted.

6 Evaluation

We evaluate the two workloads that we have implemented. We describe our methodologies and results for B+ Tree in 6.1 and Sorting in 6.2.

6.1 B+ Tree

Baseline: Our workload consists of each client making 100,000 random search queries on a B+Tree containing 1M keys. For our baseline, we evaluate the performance of a local-memory B+tree with the performance of a remote-memory variant. We do this to understand the cost inherent to using remote-memory. Fig. 4 shows that local memory is almost 3 orders of magnitude faster than remote memory. This is expected since remote-memory has a minimum latency of 2μs as opposed to DRAM latency of 100ns. However, we find that both local and remote implementations suffer equally as the size of each node grows. This is because such a configuration involves a large amount of data copy, further decreasing the performance.

A key takeaway from Fig. 4 is that our optimizations effectively halts the effect of larger node sizes. At 1KB value sizes, the performance of an RDMA-optimized B+tree is almost double than that of a naive remote-memory implementation.

Scalability: Fig. 5 shows the overall query throughput as concurrent clients are increased,

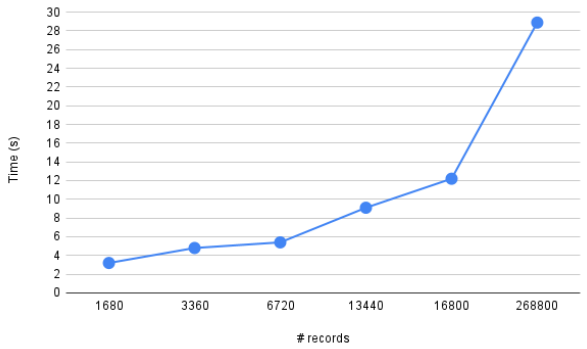

Figure 6: Constant local memory size (42000) with variable remote memory sort performance.

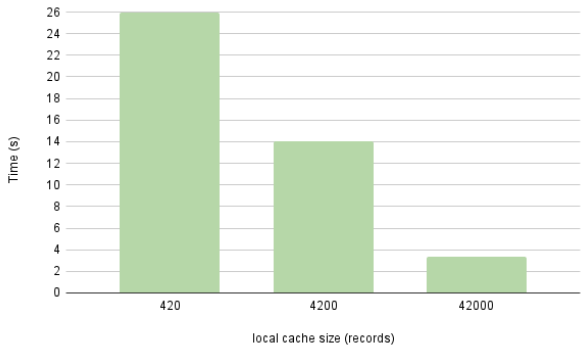

Figure 7: Remote memory has 168000 records. Sort time with variable local memory size.

where each client makes random 100,000 queries. We find that our implementation exhibits near perfect strong scaling, and neither the network nor the remote NIC become the bottleneck. This is encouraging for far-memory read-only workloads which make multiple round trips but involve minimal data transfer.

6.2 Sorting

We limited our experiments to a single machine due to time constraints although our implementation supports multiple nodes. Since the machine had only 20 cores and as our algorithm required number of $RUN/2$ clients we limited our experiments to a small number of keys for now. The data set is OF 100b records where 10B is key and 90B is value both are in ASCI. First, we evaluated the impact of a constant client memory size with variable remote memory size as shown in Fig 6. As the records sizes increases the demand for number of clients also increase for example sorting 16800 records required 40 clients whereas 26880 required 640 clients. Both the total amount of IO to disk and total number of bytes transferred via network remain same for each phase. However, increase in size increase the number of phases attributing to more time and also increases the demand for clients which results in over subscription.

Fig 7 highlights the benefits of having larger client memory size. This reduces the number of phases and in turn the number of network transfers. For example, in this case with local memory holding only 420 records will need to perform 8 phases and needs to spin 400 clients during run generation. On the other hand if client could store 42000 records, we would only need 4 clients to generate the runs and just 2 phases to produce the final output. This also indicates that the rdma bandwidth is not the limiting factor for these sizes.

7 Related work

An important line of work related to this paper are new database designs based on RDMA [4, 9, 13, 15, 5]. While [5] identified distributed indexes as a challenge in their work, it is not the main focus of their paper and was only considered as an afterthought. Furthermore, there exists other database systems that separate storage from compute [2, 8] but they dont discuss index designs based on RDMA. Another interesting work that separates compute from storage [15], however, discusses indexes stored in remote hosts that is retrieved over the network, but their assumption is that the whole index can be stored on the client memory. Our work, however, only accesses the nodes of the index that need to be accessed. Two other systems[9, 13] that focus on distributed databases built on RDMA, talk about optimizing key-value stores over RDMA and optimizing RPCs over two-sided RDMA respectively. While, we onlny work with one-sided RDMA, out datastructure and workload characteristics are completely different from the ones discussed in these papers. While external sort algo-

rithm [6] is extremely popular, and there have been works [11] that evaluate different sorting algorithms on PM, we couldn't find an evaluation of external sort using RDMA over PM.

8 Future work

1. We intend to incorporate multi-way merging per client (at the cost of added client side memory) and evaluate it for its reduced parallel connection overhead.

2. We are currently exploring in-place sorting techniques to reduce use of the auxiliary data buffers at both client and server side.

3. Our current API only allows a single in-flight memory transfer operation. We would like to extend our API to support multiple in-flight requests. We believe our current architecture will allow us to easily add this.

4. Once we are comfortable with building RDMA systems, we would love to tackle the workloads we outlined in §3.

9 Contributions

Everyone contributed equally to all aspects of the project.

References

[1] Available first on google cloud: Intel optane dc persistent memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[2] Building a database on s3, howpublished=.

[3] Configuring nvme. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/nvme-over-fabrics-using-rdma_managing-storage-devices#configuring-an-nvme-rdma-client_nvme-over-fabrics-using-rdma.

[4] Designing databases for future high-performance networks. http://sites.computer.org/debull/A17mar/p15.pdf.

[5] Distributed transactions at scale, howpublished=.

[6] External sort, howpublished=.

[7] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi. Microsecond consensus for microsecond applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 599–616, 2020.

[8] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 1021–1024, New York, NY, USA, 2010. Association for Computing Machinery.

[9] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[10] E. Frachtenberg. Holistic datacenter design in the open compute project. volume 45, pages 83–85, 2012.

[11] Y. Hua, K. Huang, S. Zheng, and L. Huang. Pmsort: An adaptive sorting engine for persistent memory. *Journal of Systems Architecture*, 120:102279, 2021.

[12] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[13] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, Nov. 2016. USENIX Association.

[14] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[15] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 663–676, New York, NY, USA, 2015. Association for Computing Machinery.

[16] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1), mar 2017.

[17] P. MacArthur and R. D. Russell. A performance study to guide rdma programming decisions. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 778–785. IEEE, 2012.

[18] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[19] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 57–69, USA, 2018. USENIX Association.

[20] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[21] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[22] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[23] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery.

[24] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY,

USA, 2019. Association for Computing Machinery.

[25] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[26] J. Xu and S. Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 323–338, 2016.

[27] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association.