Data Pipes: Declarative Control over Data Movement

Lukas Vogel
Technische Universität München
lukas.vogel@in.tum.de

Daniel Ritter
SAP, HPI
daniel.ritter@sap.com

Danica Porobic
Oracle
danica.porobic@oracle.com

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Alberto Lerner
University of Fribourg
alberto.lerner@unifr.ch

ABSTRACT

Today's storage landscape offers a deep and heterogeneous stack of technologies that promises to meet even the most demanding data-intensive workload needs. The diversity of technologies, however, presents a challenge. Parts of it are not controlled directly by the application, e.g., the cache layers, and the parts that are controlled, often require the programmer to deal with very different transfer mechanisms, such as disk and network APIs. Combining these different abstractions properly requires great skill, and even so, expert-written programs can lead to sub-optimal utilization of the storage stack and present performance unpredictability.

In this paper, we propose to combat these issues with a new programming abstraction called *Data Pipes.* Data pipes offer a new API that can express data transfers uniformly, irrespective of the source and destination data placements. By doing so, they can orchestrate how data moves over the different layers of the storage stack explicitly and fluidly. We suggest a preliminary implementation of Data Pipes that relies mainly on existing hardware primitives to implement data movements. We evaluate this implementation experimentally and comment on how a full version of Data Pipes could be brought to fruition.

1 INTRODUCTION

The storage hierarchy in state-of-the-art computing systems has become deeper and more heterogeneous. Besides the traditional cache layers and DRAM, persistent memory (PMem) is now available on Intel platforms [52], and soon High-Bandwidth Memory (HBM) will also be on the market [45] [1]. Technologies such as RDMA-enabled networks, which are now very common, can connect stacks from different machines with low latency [21, 46]. In addition, platforms that enable computational storage and near-data processing are becoming more widely available [11, 30]. Programming with such a diverse set of technologies requires quite a skill set. Such a skill set is much more crucial in data-intensive systems, where efficient data movement is paramount.

The challenges to obtaining efficient and predictable data movement are numerous. The following is a non-exhaustive list: (1) HDDs, SSDs, PMem, and DRAM all have different device characteristics requiring the programmers to adopt different system optimizations. (2) There are various interfaces of different granularity when accessing these devices (e. g., block, zone, key-value, byte). (3) Different workloads (e. g., OLTP, OLAP, Machine Learning) exhibit different data access patterns and require different hardware-conscious optimizations. (4) The cloud and high-performance computing infrastructure have disaggregated storage, adding network-induced unpredictability. (5) Different CPU vendors offer support for different storage, such as Intel historically supporting persistent memory while AMD supports more PCIe lanes.

The conventional wisdom in writing data-intensive systems is to deal with each storage type individually, using whichever OS, file system, or library API that is available. This approach is valid when we store large data structures but do not move much data, e.g., indexes. The programmer decides which storage layer data should reside on and creates efficient access methods. This permits targeting optimizations for particular storage device [10, 12, 32, 40, 44, 50], stack [19, 22, 29, 34], or primitive [26, 43].

This approach breaks down in scenarios where large amounts of data need to be moved. One such example is an external sort. Sort is arguably one of the most fundamental operations in data-intensive systems and, as we will show shortly, one that can significantly benefit from different storage technologies. The reason is that implementing all the data movements that an external sort requires is difficult precisely because of the diversity of storage options. The programmer is exposed directly to all the individual idiosyncrasies of each layer. We will use sort as an example throughout the paper, but many other operators and patterns exist in data-intensive systems that can benefit from significant data movement across storage layers. Rather than exposing the programmer to a jamboree of APIs and specific behaviors, we propose to give her an abstraction that can move data across any layers efficiently. We call this abstraction *Data Pipes.*

Data pipes is a *holistic*, *top-down* approach to creating data-intensive systems that utilize modern storage stacks. More specifically, data pipes offer programmers a *declarative* interface for *explicitly* controlling how data moves from one layer of the storage stack to another. Underneath, a framework determines which primitives to use based on the available hardware and requested data path. In other words, a data pipe will resort to a hardware-assisted data movement instead of wasting precious CPU cycles with `load` and `store` instructions whenever such a hardware unit exists. For instance, modern CPUs offer a little-known *uncore* DMA unit called I/OAT [5]. The I/OAT unit can move data from PMem to DRAM and

[1] Even though Intel decided to discontinue PMem, the trend of having more diversity in storage layers remains.

13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

vice-versa. For another instance, data pipes will resort to optimizations, such as DDIO [4], to move data directly to caches, skipping DRAM whenever possible. The main goal of the framework underneath the Data Pipes abstraction is to minimize data movement traffic and latency.

In summary, the contributions of this paper are the following:

- We survey the primitives that give us more control for orchestrating data movement over the storage hierarchy (Section 2).
- We demonstrate the potential of these primitives over the use case of an external sort in terms of minimizing data movement latency and traffic (Section 3) and quantify the performance of two primitives, I/OAT and DDIO, in this context (Section 4).
- We present our vision of *Data Pipes*, a *top-down* and *holistic* approach to creating a *declarative* control plane over data movement in data-intensive systems (Section 5) and discuss the guiding principles behind their design (Section 6).
- We identify a research agenda targeting different system layers (from hardware to applications) to better support the data pipes vision (Section 7).

2 BACKGROUND AND MOTIVATION

We have mentioned several storage layers but have yet to describe them. Fig. 1 depicts the typical stack in a modern computing system. The right side shows the layers as a classic *size vs. latency* pyramid. The lower storage layers are larger and slower, while the upper layers are scarcer but present better latency and bandwidth. On the left side of the figure, we present a partial list of mechanisms that allow a programmer to directly or indirectly move data between two layers. Note that the mechanisms range from complete protocols such as NVMe [38], to technologies such as DDIO, all the way to CPU instructions.

We classify the mechanisms by color-coding them according to how easy to use and accurate they are from a programming point of view. The green mechanisms are effortless for a programmer to access it—e. g., via I/O system calls—and can issue precise data transfers. One such example is the NVMe protocol. On the other end of the spectrum, we portray highly specialized mechanisms in red. They require low-level programming skills and are sometimes advisory. Examples of such mechanisms are DDIO and I/OAT. In between, there are a growing number of instructions that a savvy programmer can use to interact with the caching storage layers. When combined, these mechanisms can move data between almost any two distinct layers of the stack.

We note how non-uniform these mechanisms are. Some are transparent, such as promoting an entire cache line as a side-effect of reading data; others are explicit, such as issuing the recent CLDEMOTE instruction to perform the opposite movement. Some are implemented as hardware instructions, as above, while others are libraries. In particular, I/OAT defies classifications. As we mentioned before, the I/OAT is a DMA unit that can move data without CPU intervention.

Some frameworks try to unify these mechanisms and make them more accessible to programmers, the most notorious one being Intel's Storage Performance Development Kit (SPDK) [7]. Indeed, we extensively used SPDK for the experiments we present in Section 4. SPDK is, however, very "opinionated" on how programs must be structured. It gives the programmer access to even the I/OAT unit but forbids her from using, for instance, established threading and many other useful libraries.

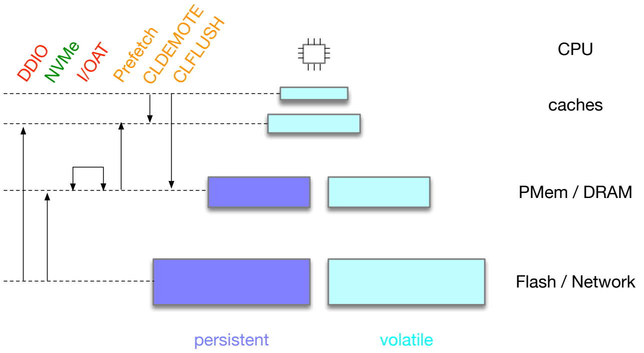


Figure 1: Data movement primitives (left) can shuffle data among storage layers (right). We claim that the set of primitives is, at best, incomplete and, arguably, incoherent from the programmer's point of view.

In contrast, we propose an API for data pipes that encapsulates the exact mechanisms SPDK does without imposing any other programming style or limitation to the application. As we commented above, we introduce these notions with the help of a motivation sorting example. Sorting is an operation we call *well-behaved* with respect to data movements. Even though we cannot predict the amount of data to move, these operations have predictable data access and movement patterns. Some other well-behaved operations and workloads in data-intensive applications are the following: *logging*, which comprises data movements of small sequential records to low-latency persistent storage; and *checkpointing*, which moves several large chunks of DRAM-resident data structures in parallel, also into stable storage, to cite a few.

The advantage of recognizing a workload as predictable—well-behaved—is that **it allows the programmer to declare the data transfers in advance**. One of the goals of Data Pipes is to give the programmer the syntax to encode these declarations. Let us see one such example in practice.

3 CASE STUDY: EXTERNAL SORT

In this section, we look at a typical external sort algorithm but concentrate on the types of data movement it generates. We pick external sort because it is a central building block for many data-intensive operations (e. g., compaction of log-structured merge trees [39], deduplication [41], and sorting query results).

An external sort is comprised of two phases: one in which data is partitioned in small batches and each batch is sorted, and one in which the sorted batches are merged. Unsorted batches move "up" the storage stack to be sorted by the CPU and then need to be moved "down" to make space for new batches. The recently sorted batches should be kept as close to the CPU as possible, as the latter will operate on them again when merging. Fig. 2 depicts the data movements for sorting (way up from storage/memory) and merging (way down). We discuss each of these movements in turn.

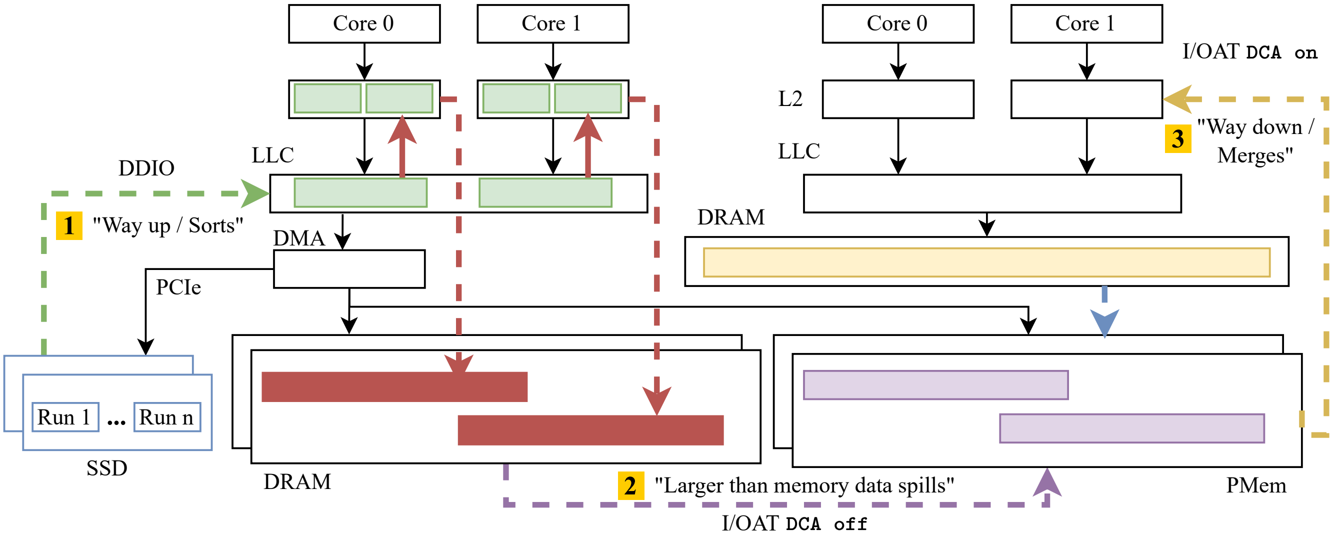


Figure 2: Hardware-efficient external merge-sort using DMA, DDIO, and I/OAT across CPU caches, DRAM, PMem, and SSDs.

3.1 Data Movements in External Sort

We assume in the following that our system contains PMem. PMem can be used in different ways, but we focus on deploying it as a staging layer between sorting and merging.

Way up/Sorts: As shown in Fig. 2, with `DDIO`, the data to be sorted 1 is read directly from persistent storage (i. e., data source) to CPU caches via the DMA engine, bypassing main memory (hence avoiding extra memory allocations). The size unit of these fetches (*runs*) can be half the LLC size per core. We need twice the space to allow double buffering in LLC rather than going to DRAM while a core is sorting the fetched data.

Larger-than-Memory Data Spills: In non-pipelined external sort, we first sort all the runs before each core merges these runs. Each core performs merging until the DRAM size is exhausted. The merge-sorted run 2 can be efficiently spilled to a staging area backed by a persistent storage device larger than DRAM, using `I/OAT` as soon as its initial block/page is produced. The staging area could use a different device than the data source (if available). Ideally, the device should exhibit low access latency, preferring PMem or the latest low-latency SSDs.

Way down/Merges: The merge-sorted runs 3 are read from the staging area, using `I/OAT` via DMA as a fetch unit similar to the way up/sorting phase. However, since the runs are already sorted, the CPU cores only need to perform a merge. This process is repeated until the run is sorted.

3.2 The Case for Hardware-Assisted Data Movement

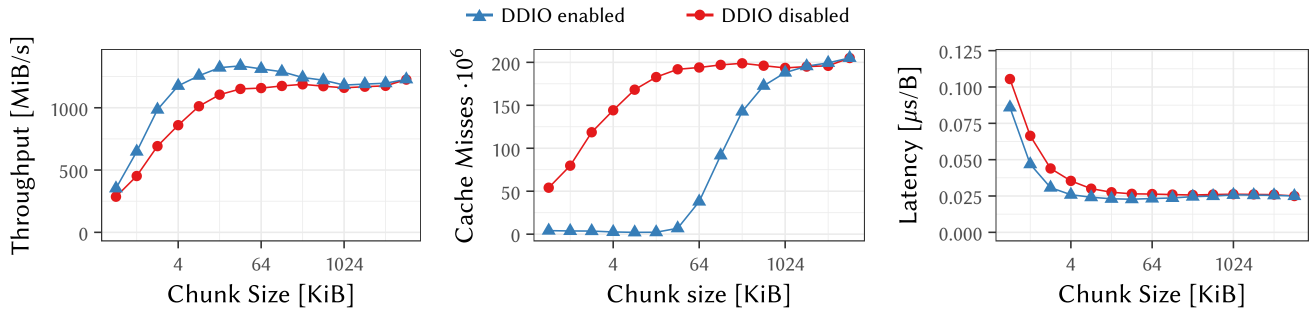
Data movements should be computationally inexpensive in theory, and in many cases, they are. Take a transfer from a modern SSD, for instance. The NVMe protocol, which fast SSDs overwhelmingly use, can be seen as a layer atop the SSD's DMA engine. NVMe allows an application to point to data it wants to write instead of moving the data itself. However, not all layers are built as NVMe, and, in practice, transfers could be CPU intensive for multiple reasons: (1) They waste CPU cycles if synchronous storage APIs are used (e. g., `read()`, `write()` system calls); (2) Moving data between different layers can incur non-trivial overheads on the side (e. g., entering kernel mode or copying buffers); (3) They are inherently expensive because of implicit aspects (e. g., networking or PMem's load/store interface requiring CPU instructions).

Besides efficiency, another observation is that some data movements may occur *explicitly* (e. g., reading a block from an NVMe SSD). In contrast, others transfers are *implicit* (e. g., the CPU flushes a cache line). Anecdotally, practitioners invest a significant amount of time trying to coerce *implicit* data movements into efficient patterns for their applications.

With Data Pipes, we aim to make all data movements explicit. One way to achieve so is to assume that DMA units, rather than the CPU, will perform all movements. As mentioned above, this is already the case when transferring data in or out of NVMe devices. It is also the case when using RDMA-capable network interface cards (NICs). For the remaining movement possibilities, we resort to the I/OAT unit as a DMA agent. In other words, we avoid using the CPU `load` and `store` instructions to transfer data whenever possible. We call this strategy *hardware-assisted data movement.* Putting it differently, one may see Data Pipes as wrappers to different DMA units available in a computer system. We discuss what Data Pipes look like shortly, but before doing so, we perform some preliminary experiments to quantify the effects of hardware-assisted data movements.

4 EXPERIMENTS

When using Data Pipes instead of coding data transfers by hand, a programmer can move the responsibility of optimizing those transfers to the Data Pipes implementation. In this section, we evaluate such potential optimizations using two hardware-based mechanisms: `DDIO` and `I/OAT`. We start with one experiment of data loading from storage into LLC/L3 with DDIO. Then, we evaluate



(a) Throughput of DDIO workload **(b) L3 Cache Misses (from `perf`)** **(c) Latency of DDIO workload**

Figure 3: Performance with enabled and disabled DDIO (one CPU core) with parallel DRAM-intensive STREAM workload.

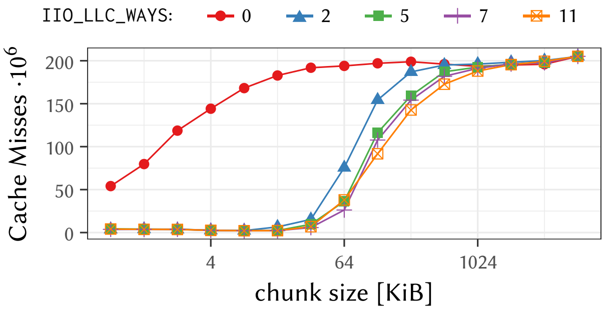


Figure 4: Impact of the `IIO_LLC_WAYS` register on cache misses with DDIO enabled.

I/OAT for data spills from memory to storage and vice versa in two separate experiments.

We use a machine in our experiments equipped with a Xeon Gold 6212U CPU with 24 physical cores, 192 GiB RAM, 768 (6 · 128 GiB) first generation PMem, and a Samsung 970 Pro (PCIe 3) SSD.

4.1 Fast Load from Storage to Compute

In our first experiment, we use the data movement accelerator DDIO, enabled by default on current Intel platforms, to move data from SSD toward the CPU for sorting. DDIO directly places data with DMA via PCIe into the L3 cache, assuming that requested data will be needed soon. Since we can also use DMA to read data from NVMe SSDs, we can re-purpose DDIO to load data from SSD and put it directly into L3 (cf. 1 in Fig. 2).

The experiment consists of issuing reads at queue depth 32 and then iterating over the read data once whenever a request is finished (forcing all data into the caches). We use SPDK on one CPU core to copy integers in increasing chunk sizes to DRAM (with DDIO disabled) or L3 cache (with DDIO enabled) and sum them up. To show that we can perform storage-to-cache movement without using DRAM bandwidth, we fully saturate the DRAM bandwidth by creating heavy artificial traffic using STREAM benchmark [35] in parallel in the background.

Fig. 3 shows the resulting performance of DDIO on this DRAM-intensive workload. In Fig. 3b, we observe that leveraging DDIO reduces cache misses (i. e., minimizing the side-effects of heavy DRAM traffic) as long as the chunk fits into L3 cache. In addition, when the memory subsystem is taxed, DDIO can increase throughput (cf. Fig. 3a) and reduce latency (cf. Fig. 3c) even for "slow" SSDs (compared to NICs, i. e., DDIOs original use case), while freeing the CPU to do other computations (e. g., sorting).

We note that DDIO can be hard to use optimally as it depends on some hidden tuning knobs: Tuning the value of the undocumented `msr` register `IIO_LLC_WAYS` as explained by Farshin et al. [20] has a significant impact for this workload, as seen in Fig. 4. Increasing its value from 2 to 11 reduces cache misses by up to 41% at chunk sizes below 1 MiB.

4.2 Fast Load from Buffer to Memory

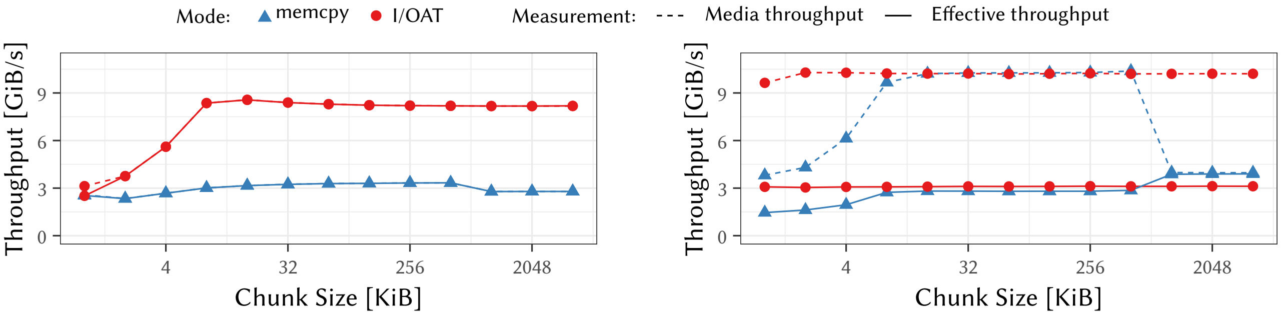
In our second experiment, we leverage I/OAT as a DMA engine to offload data movement between PMem (i. e., storage) and DRAM. This movement optimization comes in handy when loading spilled data during sorting (cf. 3 in Fig. 2). Here, offloading data movement is especially valuable since PMem uses a `load()`/`store()` interface like DRAM, where each access is a CPU instruction. This experiment uses SPDK's `accel_fw` [7] feature on a single core to issue copy requests of increasing size from PMem at queue depth 8, comparing `memcpy` (internally using non-temporal load, store) to the I/OAT backend.

The resulting throughput is shown in Fig. 5a. When deploying I/OAT, moving data from PMem to DRAM is up to ≈ 2.57× faster compared to `memcpy` on a single core. Hence, for a throughput comparable to I/OAT, three CPU cores need to be dedicated to data movement. This emphasizes the benefit of offloading data movement from the CPU.

4.3 Lack of Control for Data Spills to Buffer

In our third experiment, we look into the reverse data movement from the experiment in Section 4.2. We use the I/OAT unit once again, this time to move data from DRAM to PMem (cf. 2 in Fig. 2).

Fig. 5b shows the resulting throughput. These experiments reveal some issues with I/OAT. While writing to PMem, I/OAT is still marginally faster at chunk sizes $\leq$ 512 KiB, but it only reaches a third of the read throughput. That is far below PMem's potential write throughput. To investigate further, we measured the PMem media throughput (i. e., the write throughput the physical DIMM actually experiences), which is ≈ 10 GiB/s, close to PMem's maximum write throughput [49]. The reason for the high write amplification



(a) Moving data from PMem to DRAM

(b) Moving data from DRAM to PMem

Figure 5: Throughput when moving data between DRAM and PMem with and without I/OAT.

was first discovered by Kalia et al. [26]: I/OAT implicitly puts data into the L3 cache when it moves it to PMem using DMA, assuming it will be processed soon. This feature is called Direct Cache Access (DCA) and is not readily usable on modern CPUs. While this is great when moving data *in* from PMem, it is a huge bottleneck when moving data *out* to PMem for two reasons: (1) It evicts other data from the L3 cache and replaces it with data not intended to be accessed (otherwise, we would not have moved it out of DRAM). (2) When the CPU finally evicts the data, it does not evict it sequentially but semi-randomly. As each cache line is 64 bytes and PMem's internal block size is 256 bytes, each cache line eviction triggers a block write, resulting in up to 4× write amplification (≈ 3.27× in our measurements).

From these experiments, we note that using I/OAT to offload memory movement to/from PMem is hard to implement in practice. What compounds the issue is that Intel does not document details about DCA or how to toggle it. Even on CPUs that nominally support disabling it, it often is not exposed in the BIOS configuration and requires error-prone fiddling with Intel's `msr` registers. In our setup, Intel does not expose the `msr` register, leaving us with no way to disable DCA. In other words, by hiding these complexities behind a Data Pipe interface, we can adapt to systems that permit the configuration and fall back to a less optimized transfer in systems that do not.

4.4 Discussion

Our experiments showed that existing hardware-assisted data movement mechanisms such as DDIO and I/OAT are beneficial for database workloads, but they also uncovered several challenges.

The main issue is how obscure some of these mechanisms are. I/OAT's DCA issue when copying to PMem and DDIO's hidden tuning knobs highlights a central problem with our current data movement primitives. These primitives provide significant speedups if use cases match the scenarios for which they were originally designed. It is possible to deviate from those scenarios but not without mixed results (i. e., DDIO is great with reducing traffic in caches but shows limited latency improvements and is hard to tune; I/OAT is hard to exploit). We require further experiments to determine all the constraints, tuning knobs, and implicit assumptions required to make the hardware mechanisms work for Data Pipes.

5 OUR VISION: DATA PIPES

We discussed Data Pipes as an abstraction and showed that some powerful hardware-assisted data movement mechanisms to support them are available, even if these mechanisms are tricky to configure. Therefore, we hide the implementation complexity under a friendlier API, and expose only concepts familiar to data-intensive programmers. We chose to do so by making Data Pipes resemble a new type of descriptor/object in a C/C++ sense. Once a pipe is instantiated, it can transfer data from source to destination via a special call. We make the pipe's source and destination explicit by introducing the concept of *resource locators*. Curiously, these decisions still allow us to experiment with different programming styles when using Data Pipes.

Data Pipe Flavors. We propose three different flavors of data pipes. They mainly differ on whether the programmer wants to: (1) wait on the transfer, (2) be asynchronously notified when it is done, (3) or whether there is OS support for the wait. For each flavor, Fig. 6 shows an example of how an application would use it. Fig. 6a implements the data movement step **1** of Fig. 2 using flavor (1), i. e., it moves data in L2 cache-sized chunks to the cache of each core, where it is sorted and then demoted to DRAM. Fig. 6b implements step **2** using flavor (2), spooling sorted runs from DRAM to the backing PMem. Fig. 6c illustrates flavor (3) by retrieving sorted runs from PMem and merging them (step **3**).

We discuss these flavors shortly but first present the abstraction of the *resource locator* in more detail.

5.1 Resource Locators

Resource Locators can declare a source or destination of a data movement. We highlight resource locators code in green. Different types of resource locators are used for different storage devices, e. g., a `DRAMResourceLocator`, an `SSDResourceLocator`, or a `CoreCacheResourceLocator` for the L2 cache of a given core. Employing this abstraction has at least two advantages. First, it presents a uniform start and endpoint to which a data pipe can connect. Second, it enforces type system of sorts on data movement. Traditionally, most data access is "loosely typed" as a pointer and an offset (memory-mapped devices) or file descriptor (block devices). With the thin abstraction of resource locators and pipes, data movements have become intentional and "strongly typed." While a locator internally might still be a pointer, the user is now

```
1  size_t buffer_sz = 1 * GB;
2  size_t run_sz =
       CoreCacheResourceLocator::
       CacheSize;
3
4  SSDResourceLocator ssd_locator("/path
       /to/ssd/file");
5  DRAMResourceLocator dram_locator(
       buffer_sz);
6
7  do_parallel_foreach_core {
8    size_t offset = core_idx * run_sz;
9
10   //Allocate cache at the local core,
       backed by a memory area
11   CoreCacheResourceLocator
       cache_locator(run_sz,
       dram_locator + offset);
12
13   Pipe ssd_uppipe(ssd_locator,
       cache_locator);
14   Pipe cache_downpipe(cache_locator,
       dram_locator);
15
16   //Will try to use DDIO, since this
       is a disk to cache transfer
17   ssd_uppipe.transfer(
18     offset /*ssd offset*/,
19     0 /*cache offset*/, run_sz);
20   sort(cache_locator.data(), run_sz);
21     // Will try to use CLDEMOTE since
         this is a cache to RAM transfer
22   cache_downpipe.transfer(
23     0 /*cache offset*/,
24     offset /*dram offset*/, run_sz);
25 }
```

(a) Straightforward, loading data from SSD and sorting it into runs. Step 1 in Fig. 2.

```
1  size_t base = 0;
2  size_t pmem_offset = 0;
3
4  PMemResourceLocator pmem_locator("/
       dev/dax0.1", pmem_offset, sz);
5
6  PipeRuntime runtime;
7  runtime.fork_and_start();
8  Pipe dram_downpipe(dram_locator,
       pmem_locator, &runtime);
9
10 while (is_sorting_runs) {
11   //Collect pending tasks
12   vector<future> futures;
13   for (size_t offset = 0; offset <
       watermark; offset += run_sz) {
14
15     promise<void> write_promise;
16     futures.push_back(write_promise.
        get_future());
17
18     //Uses I/OAT with disabled DCA to
         not pollute the cache. Moves
         are scheduled asynchronously by
         the runtime, the promise is
         tranferred to the runtime.
19     dram_downpipe.transfer_with_cb(
20         offset /*dram offset*/,
21         base+offset /*pmem offset*/,
22         run_sz, move(write_promise));
23   }
24   base += watermark;
25   // Block until all moves are done
26   wait_all(futures).wait();
27   futures.clear();
28 }
```

(b) Inversion of Control, store sorted runs on PMem. Step 2 in Fig. 2.

```
1  int k = 4;
2  size_t merge_sz = k * run_sz;
3
4  int pmem_uppipe_fd = create_pipe(
5    pmem_locator,
6    cache_locator);
7
8  int epoll_fd = epoll_create1(0);
9  epoll_event pmem_pipe_op;
10 pmem_pipe_op.events = EPOLLTRANSFER;
11 pmem_pipe_op.fd = pmem_uppipe_fd;
12 epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0,
        &pmem_pipe_op);
13
14 do_parallel_for_each_core {
15   // Wait until the pipe can make the
        transfer as the I/OAT unit
       might be occupied elsewhere.
16   epoll_event event;
17   epoll_wait(epoll_fd, &event, 1);
18   if (/* error */)
19     break;
20
21   size_t offset = core_idx*merge_sz;
22
23   // Issue a DMA request from PMem
       using I/OAT. Recall that the
       cache locator is backed by a
       memory area
24   pipe_transfer(
25     pmem_uppipe_fd,
26     offset /*pmem offset*/,
27     0 /*cache offset*/,
28     merge_sz);
29   merge(k, event.locator);
30 }
```

(c) OS supported, load and merge sorted runs. Step 3 in Fig. 2.

Figure 6: Three flavors of data pipes and how to employ them in the external sort example.

forced to think about what that pointer represents and where this data is supposed to be moved to, which aligns with our earlier goal of making data movement *explicit* and *declarative*.

Underneath each locator type, we include code that makes that storage area available for use. The locator is responsible for acquiring/releasing that resource (e. g., malloc()/ free() for DRAM, issuing NVMe commands, through io_uring for instance, for SSDs). The design of each locator thus depends on the resource it manages:

- Locators backed by a file take the path to a file as an argument (e. g., the SSDResourceLocator in Fig. 6a, Line 4)
- A DRAM locator only needs a size to be constructed (Fig. 6a, Line 5) as it allocates its memory by itself.
- Locators not referencing addressable memory, such as the core cache locator that references L2 cache (Fig. 6a, Line 11), need to be backed by a memory area.

5.2 Data Pipes

A Data Pipe connects two resource locators *A* and *B*. For simplicity, it is unidirectional, so it can only transmit data from *A* to *B*. As Fig. 6 shows, all flavors are declarative.

To use a Data Pipe, a programmer first declares the locators she intends to use. She then prepares data movement by connecting the locators with pipes before moving data and performing additional computations. This concept can be rendered in different ways. We describe three variations in Fig. 6 next.

A straightforward flavor. Here (Fig. 6a), pipes are objects initialized with source and target resource locators (Lines 13 – 14) and provide a transmit() method. This method takes two offsets, one for the source and one for the target resource locator. The transmit() call blocks until data is successfully moved. This flavor is easy to implement, e. g., as a library, and integrate into an existing application as the caller of the pipes never relinquishes control (similar to traditional blocking I/O).

Inversion of control flavor. This flavor (Fig. 6b) schedules data movement asynchronously. A runtime, initialized once (Lines 6 – 8), runs concurrently with the application and is responsible for scheduling. The pipe's transmit() method is asynchronous and signals their completion via a future argument (lines 15–22).[2] The application thus relinquishes control to the pipe runtime.

The advantage of this approach is that multiple data pipes can run in parallel with a central coordinator keeping track of progress and scheduling data movements optimally. In this example, a single thread can trigger multiple data movements in parallel, leaving scheduling and CPU allocation to the pipe runtime. The downside is that inversion of control is often hard to incorporate into an existing application as it might complicate the programming model and add synchronization overhead. Communicating over a future involves a mutex which might add negligible overhead when moving megabytes of data but is very expensive if moving just a few bytes between caches.

An OS-supported flavor. The previous approaches depend on a library written in the application's language. In this third flavor (Fig. 6c), a pipe is an abstraction at the OS level:[3] They are OS concepts represented by file descriptors (lines 4 – 6), and so one can `transfer` data (lines 23 – 28) analogous to `pread/pwrite`. Leveraging OS support, for example `epoll` in Linux, the application can monitor the pipe's state. As shown by lines 8–12, the application uses `epoll` to obtain a file descriptor that allows it to get notified whenever the pipe can start a new transfer. Afterward, we spawn multiple threads that wait until the pipe is ready to accept new requests (lines 15 - 19) via the `epoll` API and then issue a transfer request.

From a programmer's perspective, this approach is an abstraction level below the other two approaches: The user has to check whether the data pipe is in the correct state before issuing any requests. While this approach is more involved than the other two, it comes with two advantages: (1) It takes a big step towards being programming language agnostic, as it relies on and extends a universally known and supported interface (`epoll`, `pread/pwrite`) of the operating system. (2) It serves as a foundation upon which a library for the other two approaches can be built: Encapsulating lines 15 – 28 into a `transfer` method yields the behavior of the "straightforward" approach, adding a request to a queue before `epolling` on a separate thread yields the "inversion of control" approach. This approach, however, also has the downside of requiring still-to-be-developed kernel support.

5.3 Data Pipes Optimization

We note that in well-behaved workloads, there are often options to where to move data. In our external sort illustrated in Fig. 2, for example, we arbitrarily decided to spill runs to PMem. However, PMem is not universally available, e. g., in AMD CPUs. Depending on the system configuration, we could decide to store sorted batches on the source SSD or even on a second SSD, if available. Since data pipes already follow a declarative approach, we can abstract over which intermediate storage device the runs are placed. We illustrate here a fourth possibility: that in which an optimizer within the runtime picks the "right" storage device during execution instead, considering the storage devices available in the system.

Fig. 7a illustrated this possibility. It depicts how multiple threads sort small runs, which are transferred via (for now) abstract data pipes to a staging area, and from there, are merged, producing the sorted output. Since data pipes are declarative, an optimizing runtime can decide during execution how to *instantiate* the abstract pipes. Fig. 7b shows two such options. A sort might start with path (a), storing sorted runs in PMem, and switching to option (b) using an SSD when PMem is exhausted. Since pipes are declarative, this would come with minimal overhead for the programmer and would allow the algorithm to be split into smaller parts that can be connected via pipes in a dynamic manner.

6 DATA PIPES PRINCIPLES

We have shown above a few examples of how data pipes could be used as a programming artifact, but we have yet to discuss the guiding principles behind their design. This section does so. We start with two principles that were already demonstrated:

(1) Declarative. The programmer declares beforehand which data pipes they wish to use, e. g., from a PCIe NVMe SSD to a cache layer or from there to PMem. The upfront declaration makes the intention of the programmer *explicit*, which (1) allows the system to make specific optimizations (e. g., enabling or disabling DCA), and (2) gives the system a way to *reject* ways to move data for which no suitable optimization is implemented. The programmer can always check if the intended data path is used and adjust the application logic otherwise.

(2) Composable. As seen in our external sort example in Section 3 and Fig. 7, data movement primitives depend on each other's results. Data is loaded, operated on, and then moved again (i. e., spilled back to background storage or moved to the cache of a different core). Making data pipes declarative allows the programmer to *compose* them and thus indicate which dependencies between computation and data movement exist. This approach is very similar to traditional query engines where optimizing data movement also plays a big part (i. e., vectorized vs. code generation).

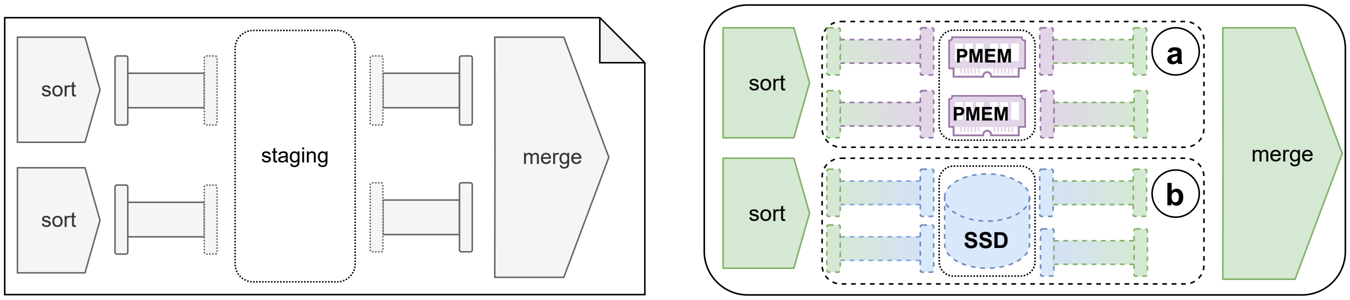
There are additional principles we now introduce that are just as integral part of data pipes proposal:

(3) Configurable. Current data movement primitives are hard to configure. They either (1) cannot be configured at all (caching behavior), (2) only be configured coarsely (e.g., globally disabling/enabling the prefetcher), or (3) rely on undocumented `msr` registers (e. g., DCA, amount of L3 cache available to DDIO). Data pipes can instead expose those tuning knobs to the programmer. This allows for tighter integration between software and hardware, as each knob can be tuned to the application's specific needs. In our DDIO experiment, tuning the value of the undocumented register `IIO_LLC_WAYS` significantly reduces cache misses as we have shown in Fig. 4. Exposing and documenting features like this would thus make it easier to benefit from DDIO.

(4) Visible State. Being configurable alone is no silver bullet: Some aspects cannot be tuned as they are fixed hardware properties (e. g., cache associativity or cache line size), and it is unrealistic to expect hardware vendors to make them tuneable. Data pipes, however, can make such aspects and their state visible to the programmer. As such, the programmer is not forced to guess or infer such constants

[2] We use C++'s promises, but C-style callbacks would work just as well.

[3] Note that this is just a proposal on how such an interface could look. The implementation would require specialized kernel support.



(a) Blueprint for a declarative external sort using pipes.

(b) Instantiated template with data pipes: Operator can be instantiated using PMem(a) or SSD as staging area (b).

Figure 7: Using data pipes to make algorithms modular.

via heuristics (e. g., CPU generation, manufacturer, benchmarking), thus making them more confident in the applicability and benefits of a particular pipe upfront.

Lastly, we have indirect goals for the data pipes API:

(5) Orthogonal to existing primitives. Data pipes do not rely on hardware vendors implementing *new* data movement primitives. While we believe that programmers would profit from additional data movement primitives that are not currently available, there is already a huge benefit in making existing primitives more easily accessible and configurable. We thus do *not* urge hardware vendors to invent a *new* ways to resolve all challenges in designing hardware-conscious data management software. We instead want them to enhance the interfaces to current data movement primitives to make it easier for applications to benefit from them.

(6) Inspiring new primitives. On the other hand, data pipes can pave the way for creating new primitives that benefit data-intensive systems, since they offer a straightforward and user-friendly way for programmers to indicate their desired data orchestration motives. This mode of indicating desired movements and paths could ease the communication between hardware vendors and software programmers to better address the needs of data-intensive systems.

It is worth mentioning that our work is aligned with other efforts to provide higher-level abstractions on which to build data-intensive systems. Some recent examples of this line of work are `DPI` [9] and `DFI` [46], which help programmers to utilize RDMA networks, and `xNVME` [33], which does the same in spirit but for fast NVMe devices. We share several traits with these abstractions but try to take a unified view of data movement independent of data location.

While data pipes can easily be used to implement well-behaved workloads efficiently, other workloads could be more challenging. For example, OLTP is characterized by the unpredictability of the read and write patterns. In literature, a common way to handle such not-well behaved workloads is to create hardware-conscious data structures such as log-structured merge trees [39, 44], B-epsilon tree [14], Plush [50], APEX [32], and others. The main design goal when creating such data structures is to morph the workload's unpredictable data access patterns or movement into a more well-behaved pattern for the target storage device. There are also recent works, such as Umzi [34], Mosaic [51], or NovaLSM [22] that target multiple layers of storage hierarchy or disaggregated storage. Enhancing these proposals with data pipes should not be an issue as long as the predictable access patterns are identifiable.

7 RELATED WORK AND RESEARCH AGENDA

It is possible to turn data pipes vision into a real-world framework even by using the available primitives and software libraries today. However, to be able to create a flexible and efficient framework across different programming languages and computer infrastructures (bare metal to the cloud), additional support from different computer systems layers is essential. We are not the only group researching such solutions. Therefore, in this section, we identify related and future research directions that can enable better support for data movement in general, and Data Pipes in particular. We divide these efforts according to context in which they are being studied: from the OS, the hardware, or from a cloud infrastructure perspective.

7.1 Operating Systems

While performing the preliminary experiments for our vision of *data pipes* in Section 4, we relied heavily on SPDK to get access to the low-level primitives to control data movement. However, this comes at the cost of using a very niche and unintuitive programming model. To keep the data pipes programming model as simple and adoptable as possible across programming languages and infrastructure deployments, it would be ideal to have better OS support for accessing low-level primitives rather than always relying on OS-bypass techniques (such as Arrakis [42], Demikernel [54], Persephoné [18]).

Furthermore, enabling an application that uses data pipes to collocate with other applications that may not use data pipes requires OS support as well. The OS needs to be aware of the data pipes and avoid swapping memory or evicting last-level cache (LLC) blocks that are explicitly needed by a pipe. One solution is to reserve part of the main memory/CPU caches to be exclusive to data pipes or to give priority to data pipes to prevent other collocated applications from thrashing memory or LLC.

These desiderata are not so far-fetched and would benefit more than data pipes, given recent efforts that already crave more explicit control over memory regions. For example, to tackle DDIO's problems, IAT [53] and IDIO [8] devise efficient frameworks that can monitor I/O and cache traffic to customize data placement in the LLC for better performance isolation. Performance isolation on LLC can also be achieved by using Intel's Cache Allocation Technology (CAT) [37] or by configuring DDIO usage via some recently discovered mechanisms [20]. Furthermore, DimmStore [28] explores different data layouts in main memory chips for energy efficiency. Differentiated storage service [36] allows to classify I/O operations to process different requests with proper policies. Data pipes can potentially leverage such differentiated storage services and LLC management techniques to use dedicated policies with priority, avoiding thrashing across layers.

7.2 Hardware

The current I/OAT unit in the Intel Xeon line of chips is an example of a DMA unit that can support transfer between "upper" layers of the storage hierarchy, such as caches, DRAM, and PMem. It has been shown that it can free the CPU while it performs asynchronous memory copies [47, 48]. This DMA unit, however, can be improved in at least three ways. First, while it delivers latency benefits over, for instance, the highly optimized glibc's `memcpy()` [16], it may present lower bandwidth when it comes to small data transfers. Second, the unit presents a limited number of channels. The exact number is a piece of information protected under NDA, but the maximum number of channels reported has been 16 [16]. For comparison, we note that the number of *tags* in a PCIe Gen 3 system, arguably the equivalent mechanism to support parallel transfers, is 256. This number keeps growing; it is 1024 for PCIe Gen 4–and both PCIe generations allow *extended tags*, which further increases this number. Third, the I/OAT unit does not support advanced transfer mechanisms such as scatter/gather [17], in which several non-contiguous memory ranges are transferred in one operation. Despite all the limitations, there have been reports of successfully incorporating I/OAT into sophisticated data movement schemes [15].

Studies have also experimented with more powerful DMA units, e.g., *memif* [31]. That work, however, confined the use of the DMA unit to the operating system's use, for instance, for data movement caused by page migration. They justify the decision by noting the lack of mechanisms to notify an application once a requested transfer is done. We demonstrated in Figure 6 three possible ways of dealing with the issue. Putting it differently, we believe that a DMA unit that overcomes the challenges we listed above can be quite useful for data pipes and can be successfully made accessible to applications.

We seek a future DMA unit with extended capabilities in another specific direction: increased reach. By increased reach, we mean accessing a portion of the storage hierarchy that remains closed. For instance, nothing can reach the CPU registers that do not come from the L1 cache. Recent examples, such as the `nanoPU` NIC [23], show that transferring data straight into the CPU registers can significantly reduce communication latency. This, in turn, can support new algorithms such as record-breaking sorting techniques [24]. Moreover, we also mean by increased reach that a more modern unit should keep pace with any new type of memory that newer systems will bring. One such imminent example, is High-Bandwidth Memory (HBM) [25]. The next generation of Intel Xeon chips, codenamed Sapphire Rapids, will support this type of memory [45], and there have been reports of the HBM benefits for the kind of data-intensive applications that we address here [27].

7.3 Cloud Infrastructure

Cloud infrastructure is becoming the de-facto environment for the development and deployment of modern applications, and we believe that data pipes have the potential to be very valuable both to cloud infrastructure providers and application developers. Currently cloud infrastructure providers offer a lot of flexibility of compute and storage deployments ranging from a wide variety of virtual machine and bare metal instances [2, 3] to fully flexible resource sizing [6] to stateless compute [1]. However, data-intensive applications often make resource sharing challenging and can easily become a noisy neighbor to others. Data pipes can alleviate this problem by making data movement predictable, thus also helping with scheduling and balancing resource usage in shared infrastructure environments.

From the cloud application perspective, the goal of predictable data movement performance often requires over-provisioning shared or provisioning dedicated infrastructure to avoid noisy neighbors. Exposing data pipes as a first-class resource with predictable throughput and latency would help ensure performance predictability at the application level. Furthermore, they can also become a flexible infrastructure building block with specific latency and throughput characteristics. Finally, using a common API for data movement across different layers of memory and storage hierarchy would make it much easier for applications to use each new and improved generation of devices without significant application changes.

8 CONCLUSION

In this paper, we motivated and illustrated a vision, *data pipes*, where the programmers can dictate and fine-tune how data moves from one storage layer to another in a declarative manner. Data pipes can make data movement more *visible* and *configurable* at the application layer. Moreover, they would not clash with existing low-level primitives to control data movement while having the potential to inspire new ones. It can allow a user-friendly abstraction while making use of the modern storage stack to achieve low latency and reduce data movement traffic in data-intensive systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Schloss Dagstuhl and the organizers and participants of the Dagstuhl Seminar 22111 on *Database Indexing and Query Processing* [13].

REFERENCES

[1] AWS Lambda. https://aws.amazon.com/lambda/. [accessed December 11, 2022].
[2] Azure Virtual Machine series. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/. [accessed December 11, 2022].
[3] Google Cloud Machine families resource and comparison guide. https://cloud.google.com/compute/docs/machine-resource. [accessed December 11, 2022].
[4] Intel Data Direct I/O Technology. https://www.intel.ca/content/www/ca/en/io/data-direct-i-o-technology.html. [accessed December 11, 2022].
[5] Intel I/O Acceleration Technology. https://www.intel.ca/content/www/ca/en/wireless-network/accel-technology.html. [accessed December 11, 2022].

[6] Oracle Cloud Infrastructure Compute Shapes. https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm. [accessed December 11, 2022].

[7] SPDK: Acceleration framework. https://spdk.io/doc/accel_fw.html. [accessed December 11, 2022].

[8] M. Alian, S. Agarwal, J. Shin, N. Patel, Y. Yuan, D. Kim, R. Wang, and N. S. Kim. IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors. In *MICRO*, pages 480–493, 2022.

[9] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. Dpi: the data processing interface for modern networks. *CIDR 2019 Online Proceedings*, page 11, 2019.

[10] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *OSDI*, 2020.

[11] A. Barbalace and J. Do. Computational Storage: Where Are We Today? In *CIDR*, pages 1–6, 2021.

[12] L. Benson, H. Makait, and T. Rabl. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *PVLDB*, 14(9):1544–1556, 2021.

[13] R. Borovica-Gajic, G. Graefe, A. W. Lee, C. Sauer, and P. Tözün. Database indexing and query processing (dagstuhl seminar 22111). *Dagstuhl Reports*, 12(3):82–96, 2022.

[14] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, pages 546–554, 2003.

[15] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message MPI communication with mpich2-nemesis. In *ICPP*, pages 462–469. IEEE Computer Society, 2009.

[16] Z. Chen, D. Li, Z. Wang, H. Liu, and Y. Tang. RAMCI: a novel asynchronous memory copying mechanism based on I/OAT. *CCF Trans. High Perform. Comput.*, 3(2):129–143, 2021.

[17] J. Corbet. The chained scatterlist API. https://lwn.net/Articles/256368/, 2007. [accessed December 11, 2022].

[18] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *SOSP*, pages 621–637. ACM, 2021.

[19] J. Ding, U. F. Minhas, B. Chandramouli, C. Wang, Y. Li, Y. Li, D. Kossmann, J. Gehrke, and T. Kraska. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD*, page 418–431, 2021.

[20] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX*, pages 673–689, 2020.

[21] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *IEEE ICDCS*, pages 553–560, 2009.

[22] H. Huang and S. Ghandeharizadeh. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *SIGMOD*, pages 749–763, 2021.

[23] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *OSDI*, pages 239–256, 2021.

[24] T. Jepsen, S. Ibanez, G. Valiant, and N. McKeown. From sand to flour: The next leap in granular computing with nanosort. *CoRR*, abs/2204.12615, 2022.

[25] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017.

[26] A. Kalia, D. G. Andersen, and M. Kaminsky. Challenges and solutions for fast remote persistent memory access. In *SoCC*, pages 105–119, 2020.

[27] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso. High bandwidth memory on fpgas: A data analytics perspective. In *FPL*, pages 1–8. IEEE, 2020.

[28] A. Karyakin and K. Salem. DimmStore: Memory Power Optimization for Database Systems. *PVLDB*, 12(11):1499–1512, jul 2019.

[29] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP*, page 756–771, 2021.

[30] A. Lerner and P. Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD*, pages 2852–2858, 2021.

[31] F. X. Lin and X. Liu. *memif*: Towards programming heterogeneous memory asynchronously. In *ASPLOS*, pages 369–383. ACM, 2016.

[32] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang. APEX: A High-Performance Learned Index on Persistent Memory. *PVLDB*, 15(3):597–610, 2021.

[33] S. A. Lund, P. Bonnet, K. B. Jensen, and J. Gonzalez. I/o interface independence with xnvme. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 108–119, 2022.

[34] C. Luo, P. Tözün, Y. Tian, R. Barber, V. Raman, and R. Sidle. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *EDBT*, pages 1–12, 2019.

[35] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. [accessed December 11, 2022].

[36] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated Storage Services. In *SOSP*, page 57–70, 2011.

[37] K. T. Nguyen. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family.

[38] NVM Express. Everything You Need to Know About the NVMe® 2.0 Specifications and New Technical Proposals, 2022.

[39] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[40] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, page 371–386, 2016.

[41] G. N. Paulley and P.-r. Larson. Exploiting Uniqueness in Query Optimization. In *CASCON*, page 804–822. IBM Press, 1993.

[42] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[43] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *SOSP*, page 392–407, 2021.

[44] S. Sarkar and M. Athanassoulis. Dissecting, Designing, and Optimizing LSM-based Data Stores. In *SIGMOD*, pages 2489–2497, 2022.

[45] G. M. Shipman, S. Swaminarayan, G. Grider, J. Lujan, and R. J. Zerr. Early Performance Results on 4th Gen Intel(R) Xeon (R) Scalable Processors with DDR and Intel(R) Xeon(R) processors, codenamed Sapphire Rapids with HBM. *CoRR*, abs/2211.05712, 2022.

[46] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. DFI: the data flow interface for high-speed networks. In *SIGMOD Conference*, pages 1825–1837. ACM, 2021.

[47] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing efficient asynchronous memory operations using hardware copy engine: A case study with I/OAT. In *IPDPS*, pages 1–8. IEEE, 2007.

[48] K. Vaidyanathan and D. K. Panda. Benefits of I/O acceleration technology (I/OAT) in clusters. In *ISPASS*, pages 220–229. IEEE Computer Society, 2007.

[49] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Building blocks for persistent memory. *VLDB J.*, 29(6):1223–1241, 2020.

[50] L. Vogel, A. van Renen, S. Imamura, J. Giceva, T. Neumann, and A. Kemper. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *PVLDB*, 15(11):2662–2675, 2022.

[51] L. Vogel, A. van Renen, S. Imamura, V. Leis, T. Neumann, and A. Kemper. Mosaic: A budget-conscious storage engine for relational database systems. *Proc. VLDB Endow.*, 13(11):2662–2675, 2020.

[52] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX FAST*, page 169–182, 2020.

[53] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim. Don't Forget the I/O When Allocating Your LLC. In *ISCA*, pages 112–125, 2021.

[54] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*, pages 195–211. ACM, 2021.