Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications

Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov
Texas A&M University, College Station, TX 77843 USA
carson@cse.tamu.edu,arman@tamu.edu,di@cse.tamu.edu,jkeech@tamu.edu,dmitri@cse.tamu.edu

Abstract

Many applications in data analytics, information retrieval, and cluster computing process huge amounts of information. The complexity of involved algorithms and massive scale of data require a programming model that can not only offer a simple abstraction for inputs larger than RAM, but also squeeze maximum performance out of the available hardware. While these are usually conflicting goals, we show that this does not have to be the case for sequentially-processed data, i.e., in *streaming* applications. We develop a set of algorithms called Vortex that force the application to generate access violations (i.e., page faults) during processing of the stream, which are transparently handled in such a way that creates an illusion of an infinite buffer that fits into a regular C/C++ pointer. This design makes Vortex by far the simplest-to-use and fastest platform for various types of streaming I/O, inter-thread data transfer, and key shuffling. We introduce several such applications – file I/O wrapper, bounded producer-consumer pipeline, vanishing array, key-partitioning engine, and novel in-place radix sort that is 3 – 4× faster than the best prior approaches.

CCS Concepts • **Software and its engineering → Virtual memory**.

Keywords Virtual memory; streaming; sorting; MapReduce

ACM Reference Format:
Carson Hanel, Arif Arman, Di Xiao, John Keech, and Dmitri Loguinov. 2020. Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378527

1 Introduction

In the big-data world, computation on large datasets often uses the dataflow model and relies on *streaming*, i.e., sequential transfer of data, between the components. In such applications, pipelines direct bulk data between a number of producer/consumer threads, which perform processing of input and send results to output. Examples include Google MapReduce [4], [17], [55], Microsoft Dryad [25], Stratosphere [1], Cloud Dataflow [20], and Apache projects Apex [2], Flink [3], Kafka [5], Samza [6], Spark [7], and Storm [8].

Unfortunately, most traditional streaming applications and software packages are not well-suited for *extreme-performance* data delivery, by which we mean rates close to the speed of the underlying hardware. In many cases, their throughput barely reaches 100 MB/s, even between threads on the same machine [26], [31], [35]. To put this in perspective, RAM bandwidth of desktop CPUs is approaching 100 GB/s and I/O speed will soon hit 50 GB/s with the release of PCIe 5.0 and 400 Gbps Ethernet. As a result, high-bandwidth C/C++ applications are in need of data-streaming abstractions that can operate at tens of gigabytes per second, far exceeding the capabilities of current platforms, but without the development cost that comes with designing custom optimized solutions for each specific application.

Scalable data computing has seen substantial developments at higher layers, where numerous open-source and proprietary projects, wrappers, and pipelining frameworks are built using standard OS primitives and iterator abstractions of various languages. Difficulty of debugging such applications, high coding effort, bloated structure, and reduced efficiency partially stem from programmer interaction with memory, which has not kept up with transitions towards data streaming and continues to offer the same *block-based* communication model of the 1950s.

Consider a long stream of data (i.e., much larger than RAM) that may be created on the fly or delivered by an I/O device. When building a new high-performance data-processing framework, for which existing solutions are inadequate, the user has to design algorithms for chunking the data, handling of records split across adjacent buffers, and enforcing proper synchronization between threads. Avoiding subtle bugs related to reads past buffer boundaries and truncation of tokens involves a large number of tedious out-of-bounds checks, while parallelization and read-ahead require additional layers of complexity. Further challenges include processing of

Algorithm 1: Block-based string search.

```
1  Function Search (char* str)
2      strLen = strlen(str); buf = new char [blockSize];
3      pos = 0; bufStart = 0;          ▷ current position in buffer/file
4      while (not end of data) do
5          size = blockSize − 1 − pos;       ▷ remaining bytes
6          bytes = ReadData (buf + pos, size);
7          buf[pos + bytes] = 0      ▷ NULL-terminate buffer
8          if ((ptr = strstr (buf, str)) != NULL) then
9              return ptr − buf + bufStart;
10         bufStart += bytes + pos − (strLen − 1);
11         pos = strLen − 1;      ▷ bytes carried over to next buffer
12         memcpy (buf, buf + blockSize − 1 − pos, pos);
```

Algorithm 2: String search on a stream buffer.

```
Thread Producer(char *buf, uint64 len)
    memset (buf, 'a', len);
    buf[len] = NULL;
Thread Consumer(char *buf)
    return strstr (buf, "hello world");
```

records larger than RAM, supporting existing libraries that assume in-memory operation, and managing input that does not have well-defined boundaries between tokens. For example, a single call to the C function `strstr`, which finds the first occurrence of a given substring, cannot be used to process an entire 10-TB stream arriving from the network into a host with 8 GB of RAM. At the same time, developing a highly optimized block-compatible version of `strstr`, as well as numerous other existing library functions, is not only expensive, but also wasteful.

As an alternative to discretizing input into blocks, we propose a C/C++ buffer abstraction called *Vortex* that allows the producer to perform unlimited sequential writes into a `char*` pointer in a way that appears *uninterrupted* to the compiler, i.e., as if the data entirely fit in RAM. Similarly, we aim for the consumer to read the same pointer sequentially and observe the written data in correct order, enabling scenarios such as `strstr` in the preceding paragraph. Because no explicit synchronization or block management is needed from the programmer, this framework is simple to use. Eliminating unnecessary compiler bottlenecks (e.g., block-boundary checking, offloading of registers to RAM) and ensuring zero-copy transfer between producer-consumer, this approach can be extremely fast, e.g., operate nearly at the speed of L3 cache in thread-to-thread communication. Armed with Vortex, streaming applications will no longer have to choose between "fast, but complex" (e.g., hand-tuned assembly) and "simple, but slow" (e.g., Hadoop [4]), which is a tradeoff often seen in practice, especially in cluster computing [39].

2 Motivation

2.1 Coding Simplicity

To understand the intuition behind our stream-based architecture, consider the task of finding the first offset at which a given substring appears in some input larger than RAM. For this example, assume that the content is arriving from another thread that supplies data on-the-fly. Under traditional *block-based* operation, there exists a consumer API, which we call `ReadData(buf,size)`, that fills a buffer of a given length and returns the number of bytes written.

Algorithm 1 shows a basic C solution, where function `strstr(p1,p2)` returns a pointer to the first occurrence of string `p2` inside `p1`. In this scenario, `strstr` encapsulates a third-party library that must see the entire record (e.g., string) in RAM before it can determine the outcome of the requested operation. To detect strings that span across adjacent blocks, Algorithm 1 must copy the last `strLen-1` bytes of each buffer to the beginning of the next logical block (Line 12), as well as perform meticulous calculation of the remaining buffer space in Line 5, the offset of this buffer from the start of the file in Line 10, and the source address for `memcpy` in Line 12. Such block-based data interaction forces the user to write code that is not only convoluted, but also error-prone, neither of which is particularly desirable.

Instead, we are interested in an interface that returns a buffer pointer that "magically" contains the entire stream, even if it is larger than RAM. This functionality is shown in Algorithm 2, where the first thread generates `len` bytes (e.g., 10 TB) of letter 'a' into the stream and the second thread searches the resulting buffer for the string "hello." This architecture removes the necessity to copy partial strings around, eliminates tedious coding, resulting in better debug time and resilience to pointer errors, and allows `strstr` to be replaced with other libraries (e.g., an HTML parser, data-analysis software) *without adapting them for block-based operation.*

Because threads are not aware of who supplies data into their stream pointer `buf` (or consumes from it), this example should be able to effortlessly accommodate data arrival from a file, network socket, or some real-time sensor device. Additional desired features of this framework include zero-copy I/O transfers (i.e., DMA into user space), configurable amounts of read-ahead, easy extendability to support diverse application needs, ability to instantiate multiple streams within a process, and minimal RAM usage.

2.2 Faster Iterator Abstractions

While jobs similar to Algorithm 1 require the user to construct complex wrappers on top of existing APIs, there are certain applications whose streams can be broken into *independent* records, each of which can be processed in isolation from the surrounding bytes. Such specialized cases allow the block-management functionality to be hidden behind an iterator interface, which can reduce the user's coding effort;

Algorithm 3: Summation producer-consumer with an iterator.

```
Thread Producer(Iterator *it, uint64 len)
    for (i = 0; i < len; i++) do    ▷ produce 10 TB
        it->Write(i);
Thread Consumer(Iterator *it, uint64 len)
    for (i = 0; i < len; i++) do    ▷ consume 10 TB
        x = it->Read(); sum += x;
```

Algorithm 4: Block-based iterator class.

```
Function Iterator::Write(int x)
    bufW[posW] = x;
    if posW == blockSize − 1 then    ▷ last item of the block?
        full.push (bufW); bufW = empty.pop (); posW = 0;
    else
        posW++;    ▷ move writer position forward
Function Iterator::Read()
    x = bufR[posR];
    if posR == blockSize − 1 then    ▷ last item of the block?
        empty.push (bufR); bufR = full.pop (); posR = 0;
    else
        posR++;    ▷ move reader position forward
    return x;
```

however, this sometimes creates a substantial performance penalty as we discuss next.

Algorithm 3 illustrates one example, where the producer writes a sequence of items (e.g., integers) into the stream and the consumer sums them up. Both threads receive the same pointer `it` to a block-based iterator class whose implementation is shown in Algorithm 4. The producer writes into block `bufW` at offset `posW` until the block overflows. It then sends the block pointer into the full queue and resets `bufW` to the next empty block. Both queues contain an internal mutex and two semaphores to ensure proper access from concurrent threads. The consumer operates using its own pair of class member variables (i.e., `bufR` and `posR`), which are placed into a different cache line from `bufW` and `posW` to prevent false sharing with the producer thread.

Even when both functions `it->Write` and `it->Read` are inlined in Algorithm 3, the compiler has no way of knowing whether the `Iterator` class variables can be modified by the program, which forces it to reload them on *each iteration of the loop*. Furthermore, it cannot predict if variable updates require global visibility, which results in every increment to `posW` and `posR` being stored back to RAM. Thus, the producer often issues at least three unnecessary loads (i.e., `bufW`, `posW`, `blockSize`) and one extra store (i.e., `posW`) per item. The consumer usually behaves similarly, i.e., loads `bufR`, `posR`, `blockSize` and stores `posR`, although some deviation is possible depending on the compiler and its optimization logic.

Modern CPUs (e.g., Intel Skylake, Coffee Lake) can issue two loads from the L1 cache and one store per cycle. It thus appears that the penalty of using iterators would be no more than two cycles per iteration; however, there are additional caveats. Because one of the three loads (i.e., `posW`) follows a previous store to the same address, there is a pipeline stall that has to wait for store-to-load forwarding to complete. This explains why in practice the iterator cost can be as high as $4-5$ cycles per item. Depending on what else the producer is doing, this may account for 50-80% of the runtime. Unfortunately, it is difficult for the user to correct this performance loss without eliminating the iterator and resorting to the tedious block-handling practices of Algorithm 1.

Algorithm 5: Summation producer-consumer on a stream buffer.

```
Thread Producer(int *buf, uint64 len)
    for (i = 0; i < len; i++) do    ▷ produce 10 TB
        buf[i] = i;
Thread Consumer(int *buf, uint64 len)
    for (i = 0; i < len; i++) do    ▷ consume 10 TB
        sum += buf[i]
```

The desired solution to this problem, shown in Algorithm 5, is to present the compiler with an uninterrupted virtual buffer that conceals stream-related variables and various housekeeping code in a completely separate location. Note that `buf` is an `int*` pointer and this is done without C++ tricks (e.g., overloading operator `[]`). Because `buf`, `len`, `i`, and `sum` are all stack-allocated, *the compiler knows that no other part of the program has a pointer to them.* This allows it to keep them in registers the entire duration of the loop, which curbs memory traffic to just one store per iteration for the producer and one load for the consumer.

2.3 Non-Counting Partitioning

The next problem that commonly arises in data processing (e.g., sorting, graph partitioning) is distribution of n input keys across K arrays, where the number of items delivered to each destination buffer is unknown a-priori. Assuming the application has an auxiliary array of size n, this task is normally solved by first performing a histogram pass over the input data to compute the number of keys that fall into each partition. With this knowledge, the second pass moves the keys into their position in the auxiliary array.

The main issue is that the histogram pass may account for 30-50% of the runtime depending on the complexity of the partitioning scheme (e.g., bit operations vs binary search over non-uniform boundaries). Thus, a stream abstraction that allows unlimited writes into a `char*` pointer can eliminate the counting pass and lead to significant performance benefits in such applications. While iterators with chain-linked blocks [43], [46] can be used for this purpose, their relatively low efficiency leaves much room for improvement.

2.4 In-Place Data Shuffling

Continuing with the partitioning example, the second drawback of the classical technique is the obligation to maintain physical memory for $2n$ items (i.e., both input and auxiliary arrays). Because each key needs to exist only in one of two places – either on input or output – this problem calls for a better solution. The desired operation of a stream buffer is to transparently free physical memory immediately after it has been read and move it to the location of the next write, which allows partitioning to run without doubling memory usage. Furthermore, after the keys have been processed, it would be ideal to copy them back into the input array without exceeding the existing memory envelope of size n, which would give rise to a new family of streaming sorters that operate *in-place*. Although block-based iterators can also partition keys using physical memory n, they cannot return the data back to the original array or rival the performance of a native memory interface, such as the one shown in Algorithm 5.

3 Virtual Memory in User Space

3.1 Paging

We start by reviewing capabilities exposed by the OS kernel to user applications, which will be needed later in the paper. Recall that each 64-bit virtual-memory address (i.e., C/C++ pointer) is converted into the corresponding physical address by means of a *page table* maintained by the OS [49]. Access to memory either absent from the page table of the executing process or lacking appropriate permissions causes *page faults*, which are hardware interrupts that invoke the kernel memory manager. We split the faults into three types – *soft*, which can be solved by remapping existing pages (e.g., on first access); *hard*, which require loading data from disk; and *exception*, which indicate some type of access violation. The last type is passed to the exception handler of the process, which by default terminates the application.

Our key observation is that virtual streams can be set up to generate controlled access violations, which are intercepted by a custom exception handler that transparently fixes the problem and allows the program to continue. Developing algorithms and techniques for doing so is our general topic. To cover both Linux and Windows, we use the terminology from Figure 1, which shows the main states of virtual memory seen by a user process. All space is initially owned by the OS and considered *free*. There is a reservation procedure (i.e., `mmap(PROT_NONE, MAP_NORESERVE|MAP_ANONYMOUS)` on Linux, `VirtualAlloc(MEM_RESERVE)` on Windows) that allows the application to set aside contiguous blocks of the virtual space in order to avoid conflicts with future allocation requests (e.g., from malloc/new). While not immediately usable, this space can later be converted to *committed-untouched*, which is done through `mprotect(PROT_READ|PROT_WRITE)` or `VirtualAlloc(MEM_COMMIT)`.

Mapping to physical pages does not occur until the program first writes to committed pages, at which time they become *committed-touched* and added to the *working set* of the process. Pages that are no longer needed can be returned to the operating system, by either keeping the reservation

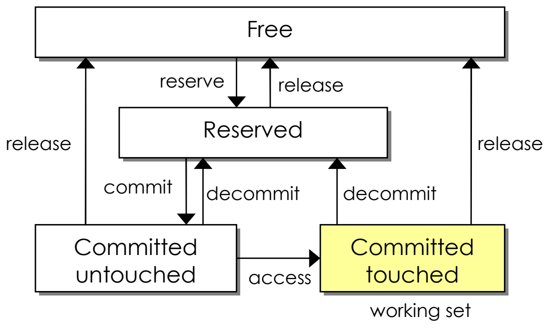


Figure 1. Taxonomy of virtual-memory states.

in place (i.e., `VirtualFree(MEM_DECOMMIT)`) or completely discarding it (i.e., munmap, `VirtualFree(MEM_RELEASE)`). In the algorithms below, we use the terms *decommit* and *release* synonymously. It should also be noted that pages in the working set can be remapped from one virtual address to another, while preserving their contents, using `mremap` (Linux) or `MapUserPhysicalPages` (Windows).

RAM usage of a process is generally measured by the size of its working set; however, there are additional costs incurred by the OS. Each page in the working set is accompanied by an 8-byte descriptor. Assuming common 4-KB pages, this yields an overhead ratio of $8/4096 = 2^{-9} \approx 0.2\%$. For reserved pages, Linux has $O(1)$ cost (i.e., the size of one node in the VMA tree), while Windows requires 1 bit for every 64 KB. Even in the latter case, this overhead is small enough (i.e., $2^{-19} \approx 0.0002\%$) to be considered negligible in many cases. For example, a 10-TB reservation on Windows requires only 20 MB of physical memory.

3.2 Exceptions

Memory-related exception handling on Linux is done by catching the `SIGSEGV` signal; on Windows, the same is accomplished by registering a callback through VEH (Vectored Exception Handling). When a user handler is invoked, the OS provides enough parameters to determine the address of the fault, the attempted operation (read/write), and the type of error (e.g., access violation, illegal instruction, stack overflow). The last piece that completes the puzzle is that the kernel suspends only the thread experiencing the exception, while letting other threads continue as normal. This makes many interesting scenarios possible, in which one thread attempts to read a block of memory that is not available yet and gets suspended in an exception, followed by another thread loading the missing block and releasing the first thread.

4 Producer Consumer

4.1 Vortex-A: Conceptualization

We start by noting that invocation of a fault handler for each 4-KB page is not likely to provide sufficient performance for memory-demanding applications. For example, Sandy Bridge

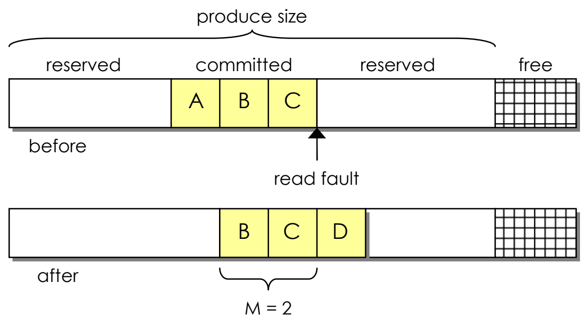


Figure 2. Operation of Vortex-A.

Intel i7 CPUs can process access violations at roughly 3M/sec, which yields 12 GB/s of throughput. Since this is well below the corresponding RAM bandwidth (e.g., 51 GB/s), it might be useful for the stream to offer additional mechanisms that control the frequency of calls to the page handler. We do this by grouping pages into *blocks* of size B (usually 1-2 MB) and issuing commit/decommit/remap on them as one unit.

Now assume existence of a producer that offers bulk data that must be processed by a consumer thread. To handle these scenarios, our first design is called *Vortex-A*. It is a simple stream that works synchronously (i.e., without read-ahead) and embeds the producer into the fault handler. This example serves as a platform for helping understand the more advanced cases in later sections. Suppose the consumer intends to read data *almost sequentially*, by which we mean that a read in block x in the buffer must be followed by reads in blocks $y \geq x - M$, where $M \geq 0$ is the number of blocks by which the program may need to come back to reprocess the already-visited bytes.

In some cases (e.g., finding the largest 8-byte integer in a stream), M might be trivially zero; however, more general situations are supported as well. In Algorithm 2, `strstr` may have to return by the length of the sought-after string. As long as `strLen` is smaller than B, usage of $M = 1$ guarantees correctness. Because Vortex is a low-level programming interface for extreme-performance applications, where memory interaction demands the highest throughout, we assume that the programmer can either eliminate repeat passes over the same bytes or at least deploy libraries/algorithms with well-defined comeback thresholds. For the majority of streaming scenarios, however, we expect either $M = 0$ or $M = 1$ to work without a glitch.

The logic behind Vortex-A is illustrated in Figure 2. The top picture shows that the producer reserves enough virtual memory to accommodate the entire stream (e.g., 1 TB on a system with 8 GB of RAM). As discussed above, this carries almost no physical-memory cost. Before providing the buffer to a user application, such as the consumer thread in Algorithm 2, the producer installs a custom exception handler whose purpose is to perform transparent page commits/decommits. In the figure, location of the next read fault is shown with an arrow after block C. Upon gaining control, the handler decommits chunk A, commits D, fills up the latter with data, and continues execution of the CPU instruction that caused the preceding exception. The working set of the process consists of $M + 1$ blocks, regardless of stream size.

The corresponding fault handler is shown in Algorithm 6, where `buf` is the start of the reserved space, `fPtr` is the address of the fault, and `dPtr` is the decommit pointer.

Algorithm 6: Vortex-A fault handler.

```
Function Handler(char *fPtr)
    dPtr = fPtr − (M+1)*B;      ▷ decommit pointer
    if dPtr ≥ buf then
        Decommit(dPtr, B);      ▷ discard block A
    Commit(fPtr, B);            ▷ commit block D
    ProduceData(fPtr, B);       ▷ write data into the block
```

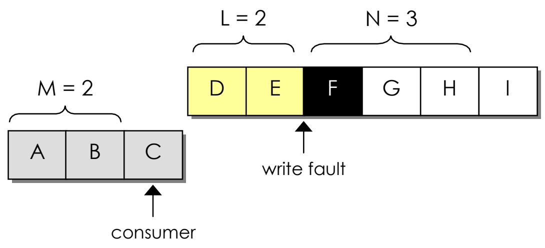


Figure 3. Consumer comeback M, producer comeback L, and read-ahead N.

4.2 Vortex-B: Virtual Data Streams

The system developed in Figure 2 is a good start; however, it has several drawbacks. First, it requires that the producer code be inserted into the fault handler, which prevents it from running concurrently with the consumer. Instead, a more desirable solution would allow function `ProduceData` to execute for the next few blocks, while the consumer is munching on the previous bytes. Second, the strength of our desired architecture is that the producer is unaware that the virtual space is split into discrete chunks. This means that it should be able to cover the entire 1-TB buffer using a single function call or CPU instruction that appears uninterrupted in user mode (e.g., `memset` in C/C++, `rep stos` in assembly). Unfortunately, the producer in Algorithm 6 is still block-based. Finally, to support *overlapped* (i.e., asynchronous) I/O with multiple pending DMA requests and other types of out-of-order writes into the virtual space, it is beneficial to allow the producer non-monotonic access to the shared buffer.

To enable the new interface, a few modifications are required. As sketched in Figure 3, let $L \geq 0$ be the size of the *producer comeback* region, given in full blocks. The producer guarantees that following a write fault in block x it never touches blocks below $x - L$. In the figure, suppose it issues $L+1 = 3$ overlapped I/O requests into blocks (D, E, F), where chunk F completes first and causes a write fault. This allows

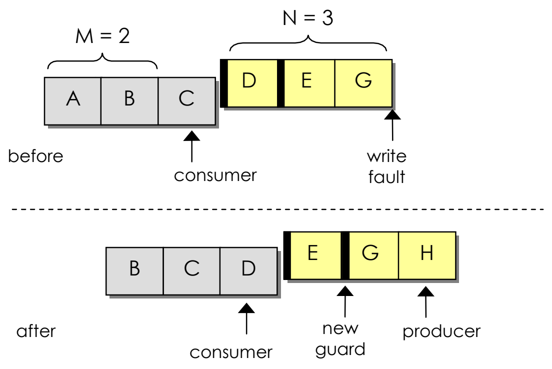


Figure 4. Operation of Vortex-B with $L = 0$.

the consumer to enter block C; however, since blocks (D, E) are still pending, the stream must stop the consumer from going any further. Additionally, let $N + L \geq 1$ be the maximum number of ready blocks that the stream can maintain. If the producer in Figure 3 is so quick that it manages to complete all five blocks $D - H$ while the consumer is still in block C, the stream needs to suspend the producer when it attempts to write into block I.

We now explain how the stream can limit the maximum distance between the producer and consumer at $N+L$ blocks. Even though write faults into reserved space give away the location of the producer, there is no such mechanism for the consumer. Our solution, which we call *Vortex-B*, is to trip up the consumer on guard pages scattered across its commit space. This is accomplished by changing the protection of the page to `PROT_NONE` in `mprotect` (Linux) or `PAGE_NOACCESS` in `VirtualProtect` (Windows). Note that the guard status does not destroy the contents of the page and can be applied to memory that is already in the working set of a process. Access violations on guard pages provide an indication to the stream where the next read is being attempted, which allows the handler to advance the producer and decommit old chunks. To make guard memory readable again, the protection is changed to `PROT_READ|PROT_WRITE` (Linux) or `PAGE_READWRITE` (Windows).

Consider an illustration on top of Figure 4, where six blocks are currently in the commit state. The producer has just finished block G and triggered a write fault at the next block. During previous write violations, the stream placed guard pages at the start of all preceding blocks. The consumer has gone though $A - C$ and removed their guards, but those at the beginning of blocks (D, E) are still in place (marked with solid rectangles).

In response to the write fault on top of Figure 4, the handler installs a guard page at the start of G and waits for the consumer to finish C. Attempts to advance into block D by the consumer spike a guard exception with a read violation. Once that happens, the stream decommits the oldest block A and commits the newest block H, allowing both producer

Algorithm 7: Vortex-B handler.

```
semFull = CreateSemaphore (0);      ▷ no full blocks
semEmpty = CreateSemaphore (N+L);   ▷ all blocks are empty
buf = ReserveMemory (size);         ▷ main buffer
Function Handler(char *fPtr, int type)
    if type == WRITE_FAULT then
        gPtr = fPtr − (L+1)B;       ▷ guard-block pointer
        if gPtr ≥ buf then
            SetGuardPage (gPtr);
            Release (semFull);      ▷ consumer can use guarded block
        Wait (semEmpty);            ▷ wait for consumer to catch up
        Commit (fPtr, B);           ▷ this block ready for producer
    else    ▷ GUARD violation
        dPtr = fPtr − (M+1)*B;      ▷ decommit pointer
        if dPtr ≥ buf then
            Decommit (dPtr, B);
        Release (semEmpty);         ▷ let producer move forward
        Wait (semFull);             ▷ wait for producer to make next block
```

Algorithm 8: Usage of Vortex-B.

```
Function main()
    len = 10 ≪ 40;         ▷ 10 TB
    Stream s (len + 1);    ▷ create a stream object
    thread p (Producer, s.GetWriterBuffer(), len);
    thread c (Consumer, s.GetReaderBuffer());
    p.join();              ▷ wait for producer to finish
    s.ProducerFinished();  ▷ releases semFull by L+1
    c.join();              ▷ wait for consumer to finish
```

and consumer to continue. This is shown at the bottom of Figure 4. Algorithm 7 implements the corresponding handler using the classical bounded producer-consumer problem [49] with two semaphores that count the number of empty and full blocks in the committed space.

The application interface of Vortex-B is given by Algorithm 8, which starts producer/consumer threads (such as those in Algorithm 2) and gives them both a `char*` pointer to work with. These threads are not aware of the underlying stream operation and treat it as a contiguous 10-TB buffer. It should also be noted that function `s.GetReaderBuffer()` must wait on `semFull` before returning to ensure that a guard has been placed in the first block of the stream *before* the consumer begins.

4.3 Vortex-C: Physical Data Streams

While Algorithms 7-8 satisfy the minimum requirements of a virtual stream, improvement is still possible in two aspects. First, Vortex-B is not particularly fast because decommitted pages are given back to the OS and the kernel must spend non-trivial amounts of work to allocate them again. This includes not only finding the best physical page for each commit request, but also memsetting that memory to zero for security reasons. Second, Vortex-B breaks if the consumer skips over guard pages and jumps forward into a partially

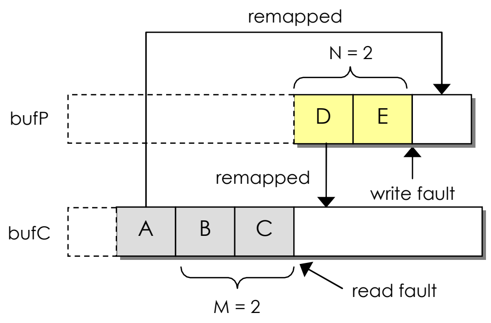


Figure 5. Operation of Vortex-C with $L = 0$.

filled block. An example would be a graph, stored using adjacency lists, in which the application needs to examine all source nodes, skipping over the neighbors.

Our immediate goal is to build a solution that removes both of these limitations. Assume that the next stream, which we call *Vortex-C*, returns two pointers – `bufP` to the producer and `bufC` to the consumer. These are virtual memory buffers, both reserved to the size of the stream. The producer attempts to write into `bufP` and causes only write faults, while the consumer reads from `bufC` and generates only read violations. The key to this architecture is quick remapping of pages from one virtual buffer to the other. This idea is illustrated in Figure 5, where the producer is stuck on a write fault after block E and the consumer just finished block C. To continue, two things must happen – chunk D gets remapped to the consumer buffer at the current read location and block A gets returned to the producer at its present write address.

Upon startup, the stream acquires a set of $N + L + (M + 1)$ blocks of physical pages that it can map to arbitrary virtual addresses. On Linux, this is done by allocating a separate buffer with `PROT_READ|PROT_WRITE` permissions and touching each page. On Windows, there is a dedicated API for this purpose (i.e., `AllocateUserPhysicalPages`). Unlike the commit/decommit approach of Vortex-B, remapping does not return memory to the OS, which in turn avoids the expensive re-allocation. Additionally, because of dual virtual buffers, Vortex-C can track the consumer during non-contiguous jumps forward into its reserve space. The underlying fault handler is the same as in Algorithm 7, except calls to commit/decommit are replaced with remapping and there is no need to set guard pages.

4.4 Multiple Streams

We now explain how different stream types can coexist within a process and share an exception handler. Our approach is to create a parent C++ class called `Stream`, from which individual Vortex-A/B/C classes are inherited. Pure-virtual functions in `Stream` provide a standardized interface (e.g., to get the read/write buffer pointer, process a page fault, obtain stream size) that does not expose the internal implementation of the inherited class. This also allows the user, if necessary, to write code for custom streams that automatically receive fault notifications related to the virtual space under their control.

In each process, we create a single instance of a special `StreamManager` class, whose job is to register as an exception handler and maintain all streams created by the application. This is done by placing the left/right boundaries $[a_i, b_i]$ of the buffer used by virtual stream i into a balanced interval tree. Given a page fault at address x, the manager performs a search for the largest $a_i \leq x$ and verifies that $x \in [a_i, b_i]$ before passing the exception to the corresponding stream s. After the stream deals with the exception, the manager tells the OS to repeat the offending instruction, which allows the program to continue. On the other hand, if no matching stream is found, the manager passes control to the next handler. Therefore, crash conditions unrelated to Vortex behave as usual, i.e., cause process termination or invoke the debugger.

4.5 Discussion

Vortex should not be viewed as a set of algorithms just limited to flavors A/B/C introduced above; instead, it encompasses a new programming paradigm that supports *fluid* memory, which appears on demand where needed and disappears after use, for various streaming producer/consumer purposes. Vortex exposes applications to memory abstractions that allow computation to directly interact with the data, disposing with traditional block-based iterators, boundary checking, bloated code needed to handle records that split across adjacent blocks, and the resulting performance loss. The strength of the proposed vision is that it is a *user-level* construct that gives anyone the flexibility to add new features, modify existing streams, and provide extensions as needed. The next section shows one such example where a hybrid of Vortex-B and C is created to support faster sorting.

5 Partitioning and Sorting

5.1 Overview

Many big-data applications rely on sorting as an intermediate step in their computation. Besides MapReduce [4], [17], other classical examples include database table joins [10], [12], [15], [18], [27], [40], [43], [51] and graph-processing tasks [33], [34], [36], [38], [37]. We define a sorting algorithm to be *in-place* if it can process inputs of size n using $n + O(1)$ space as $n \to \infty$ and return the keys back into the original array. The $O(1)$ term may contain various counters, pointers, and partially used pages that the OS has given to the application. If a method requires $2n + O(1)$ memory, we call it *out-of-place*. In general, in-place algorithms are preferred as they consume less RAM for a given input size and finish in fewer I/O passes when operating in external-memory.

Assume input consisting of n random keys of size $l = 64$ bits, which is a common scenario of interest [15], [29], [43].

On uniform integers, radix sort has a significant speed advantage over comparison-based methods (e.g., quick sort, merge sort). It will thus be our focus here. The order of bit processing in radix sort can be either MSB (most-significant bit first) or LSB (least-significant bit first). If $b \geq 1$ bits are examined at each step and the input is uniform, MSB terminates after $\lceil \log_2(n)/b \rceil$ scans over the data, where $\log_2 n$ is the number of upper bits that are sufficient for differentiation between input keys. On the other hand, LSB always needs $\lceil l/b \rceil$ iterations, which makes it less desirable for 64-bit workloads.

For this reason, our approach builds upon MSB radix sort. Its classical version [28] starts by distributing keys into $K = 2^b$ buckets, which is done by examining the upper b bits and directing each key x into bucket $x \gg (l - b)$, where $\gg$ denotes bitwise shifting to the right. Once this is done, each bucket is processed recursively in DFS order, except extraction of bucket bits at depth k now requires shifting by $l - bk$ and a bitwise AND with $2^b - 1$ to remove the irrelevant upper bits. While the idea is pretty straightforward, there are two caveats.

First, the in-place MSB version [11], [14], [15], [19], [21], [43] swaps elements at random distances far greater than cache size, which makes its memory-access pattern pretty slow and causes the method to be *unstable* (i.e., not preserve the original order between duplicate keys). In contrast, if two arrays are allocated (i.e., doubling RAM usage to $2n$), the out-of-place MSB version, also known as *bucket sort*, can stream all keys sequentially, achieve much faster rates due to better locality of reference, and maintain stability [23], [29], [30], [43], [44], [50]. As a result, MSB radix sort offers a well-defined tradeoff between speed and space.

Second, since the size of each bucket is unknown a-priori, all flavors of radix sort must use *two* passes over each key per level of recursion, i.e., once to precompute bucket lengths and once to distribute the keys. An ideal algorithm would eliminate the histogram pass and combine the best features of both in-place/out-of-place MSB versions, i.e., avoid random access (i.e., use streaming), offer an option to be stable, and keep close to n elements in the working set. As we discuss next, Vortex enables us to do just that.

5.2 Main Idea

Algorithm 9 shows pseudocode for a single-pass bucket sort. Line 1 declares an array of pointers `bucket`. Its first dimension $D = \lceil l/b \rceil$ is the maximum depth of recursion and its second dimension $K = 2^b$ is the number of available buckets. During key distribution at level i, pointer `bucket[i][j]` specifies the address in bucket j that will receive the next key. In each recursive call, Lines 3-4 set up an array of pointers `p` to provide bucket addresses for the current level and `pNext` for the next level.

Since each bucket is assumed to have enough space to accommodate all levels of recursion, newly partitioned data

Algorithm 9: Elementary single-pass bucket sort.

```
1  uint64 *bucket [D][K];
2  Function BucketSort(uint64 *buf, uint64 size, int shift, int level)
3      uint64 **p = &bucket[level][0];      ▷ pointers for current level
4      uint64 **pNext = p + K;      ▷ pointers for next level
5      memcpy (pNext, p, sizeof (uint64*) * K);
6      for (i = 0; i < size; i++)  do
7          idx = (buf [i] ≫ shift) & mask;      ▷ get bucket index
8          *pNext [idx] ++ = buf [i];      ▷ write key into bucket
9      for (j = 0; j < K; j++) do
10         sizeNext = pNext [j] − p [j];      ▷ size of bucket j
11         if sizeNext > 32 then
12             BucketSort (p [j], sizeNext, shift − b, level + 1);
13         else
14             (*sn[sizeNext])(p [j]);      ▷ call sorting network
15             memcpy (output, p [j], sizeof(uint64) * sizeNext);
16             output += sizeNext;
```

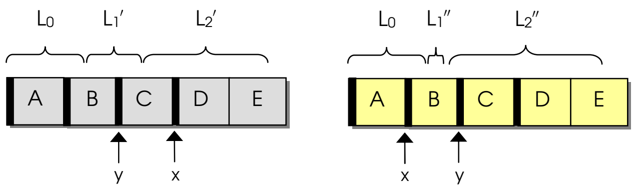
is appended to the back of each existing bucket (Lines 6-8), allowing it to expand and shrink as needed. After bucket size drops below 32, we switch to a sorting network, which is accessed by the corresponding function pointer in Line 14. If the sort needs to be stable, this call should be replaced with insertion sort.

The main difficulty with using Algorithm 9 is that it requires pre-allocating each bucket to the size of input times the number of levels of recursion. This worst-case scenario happens when all keys get appended back to the same bucket (e.g., input consisting of all zeros). To overcome this problem, our observation is that each bucket can be supported by a separate Vortex stream and reserved to the maximum size nD, which carries little cost. More importantly, reads in Line 7 can be treated by our framework as those coming from a consumer and writes in Line 8 as those from a producer. As long as memory pages released by the former can be mapped to locations needed by the latter, the algorithm can operate without needing any additional blocks from the OS beyond those obtained to support the first level of recursion.

While this framework sounds good in theory, there are open issues. They include design of a new stream that can support randomly interleaved reads/writes at each bucket from Algorithm 9 and sharing of memory blocks between different streams. The rest of the section addresses them.

5.3 Vortex-S: Sorting Streams

We start by examining why the streams developed earlier are unsuitable for sorting. Vortex-A/B rely on the OS to manage the pages, which runs into significant performance limitations, especially considering the speed at which Algorithm 9 can operate. Vortex-C is quite appealing on the surface; however, closer inspection shows that its usage of dual virtual buffers makes bucket sort highly inefficient. At deeper levels of recursion, Algorithm 9 may write to and read from



L0
L1′
L2′
A
B
C
D
E
y
x
(a) read faults in reverse order
L0
L1″
L2″
A
B
C
D
E
x
y
(b) interleaved read/write faults

Figure 6. Tricky read faults in Vortex-S.

the same block a number of times in short succession. Because each access may involve just a few keys, performing a full block remap wastes a lot of CPU cycles, which makes Vortex-C slower than Vortex-B in this application.

Instead, our next design is *Vortex-S*, which uses the general architecture of Vortex-B combined with physical pages. The new stream employs a single virtual buffer shared by the consumer/producer and relies on guard pages to trigger read faults. But because both producer/consumer in Algorithm 9 are in the same thread, Vortex-S does not need semaphores from Algorithm 7. Furthermore, instead of decommiting blocks back to the OS, Vortex-S unmaps them and returns their pointer into a special queue that is shared across all buckets. When written keys cause expansion of a bucket, its stream draws free blocks from the same queue and maps them to the location needed (more on this below).

What is peculiar about Vortex-S is that not every read fault allows deallocation of the preceding block. First, consider the scenario in Figure 6(a), where a particular bucket j consists of five blocks $A - E$. In this context, set L_0 refers to the keys written into bucket j at recursion level zero and L'_k for $k \geq 1$ indicates the keys that were generated at level k while processing some other bucket $i < j$. In the figure, Algorithm 9 writes L_0 and moves on to process buckets before j. During that time, it appends L'_1 and L'_2 to bucket j, which sets up guard pages in blocks $B - D$.

Since bucket sort is a variation of depth-first search, it unwinds recursion by first processing L'_2. This generates the first read fault of bucket j at address x; however, freeing block C, as would be done by Vortex-B, deletes parts of set L'_1 and makes Algorithm 9 fail. Continuing, notice that after reading L'_2, bucket sort eventually reads L'_1, causing the second read fault of this stream at location y. Again, freeing the preceding block B discards parts of yet-unprocessed L_0. Both issues can be handled by allowing unmapping only if the current read fault in a given bucket stream is exactly one block ahead of the immediately preceding read fault, i.e., $y = x + B$.

Figure 6(b) shows the second scenario that needs special handling. Assume that Algorithm 9 writes L_0 into bucket j and goes to read buckets $i < j$ at level zero. This produces several writes into bucket j, which at some point set up the guard page in block B. Eventually, all of the appended keys after L_0 are removed, which does not affect the guard in B, and bucket sort comes back to read L_0. This generates a read fault in position x, which is later followed by processing of some bucket $i > j$ at level 0. Its split causes L''_1 and subsequently L''_2 to be appended to the current bucket j, as shown in the figure. This creates guard pages in blocks C and D. Next, when Algorithm 9 goes to read L''_2, it generates a read fault at y, which is exactly one block beyond x. Since $y = x + B$, the rule developed in the previous paragraph allows deallocation of block B, which in turn destroys L''_1.

We can now combine insight from both cases in Figure 6 to construct a robust set of unmapping rules in Vortex-S. Suppose each stream stores internally the address x of the last read fault. Then, the next read fault at y unmaps the preceding block iff a) y is one block larger than x; and b) the guard page at y was set up *before* the read fault at x occurred. Both conditions are easily implemented in the fault handler.

5.4 Stream Pools and Free Blocks

It should be noted that Vortex-A/B/C streams generally do not need to share free blocks between each other. In Vortex-S, however, it is imperative that they do; otherwise, the sort is not in-place. To handle this functionality, the application creates a `StreamPool` object that allocates physical pages from the OS and keeps them in a queue whose pointer is provided to the constructor of all Vortex-S streams participating in the pool. On write faults, streams draw blocks of size B from the shared queue; on read faults that satisfy conditions in Section 5.3, they return the blocks back into the queue.

Upon initialization, `StreamPool` obtains enough blocks to cover input size n, plus an additional 2^b blocks. Over-allocation is necessary because the last block of each stream is only partially written to. Thus, `StreamPool` wastes $B2^b$ bytes of memory. To maintain reasonable sort speed in practice, both b and B must be constants, i.e., independent of n. Thus, the extra blocks consume $O(1)$ space as $n \longrightarrow \infty$.

5.5 Optimizations

Algorithm 9 can be deployed over Vortex-S without modification; however, for it to be competitive against the fastest alternatives certain optimizations are needed. The first two methods we use are prefetch hints to the CPU and software write-combining with non-temporal stores [10], [43], [47], [53]. The third optimization implements sorting networks using *non-branching* SWAP macros that translate into conditional move instructions in assembly. This solution, originally proposed on stackoverflow [32], is currently the fastest method for sorting small arrays. Note that at least one recent paper, RADULS2 [29], uses all three techniques.

To make a separate contribution towards faster sorting of uniform keys, we offer a novel approach to deciding the fanout at each level. We have observed that the highest speed, regardless of the CPU in our tests, was achieved when the sorting network processed on average $2^r = 8$ elements. Using more keys makes sorting networks slow; at the same

Table 1. Available Configurations

	c_1	c_2	c_3
i7 CPU	3930K (Q4'11)	4930K (Q3'13)	7820X (Q2'17)
Platform	Sandy Bridge-E	Ivy Bridge-E	Skylake-X
Cores	6	6	8
Turbo clock	3.8 GHz	3.9 GHz	4.7 GHz
RAM	32 GB	32 GB	32 GB
RAM type	DDR3-2400	DDR3-2400	DDR4-3000
Test drive	24-disk RAID	24-disk RAID	M.2 SSD
Primary OS	Server 2008 R2	Server 2008 R2	Server 2016
Secondary OS	Ubuntu 17.10	–	Ubuntu 18.10

time, aiming for less than 8 bottlenecks the program on the function pointer in Line 14 of Algorithm 9. Since this call cannot be inlined, a great deal of overhead can be saved by running the last level of recursion at the optimal value.

This requires modifying Algorithm 9 to accept a sequence of *heterogeneous* bit splits $(b_0, \dots, b_{d-1})$ and run bucket sort to depth d, where level i uses 2^{b_i} buckets. We define a combination of bit counts $(b_0, \dots, b_{d-1})$ as *admissible* for a given input size $n = 2^k$ if $k - \sum_{i=0}^{d-1} b_i = r$. For example, $n = 2^{23}$ can be processed using 10+10 bits (two levels of recursion), 8+8+4 or 7+7+6 (three levels), or 5+5+5+5 (four levels). If $s(x)$ is the speed at which the CPU can partition keys into 2^x buckets, the *optimal* solution $(b_0, \dots, b_{d-1})$ minimizes $\sum_{i=0}^{d-1} 1/s(b_i)$, subject to the constraint mentioned above. When function $s(x)$ is not available, our bucket sort selects the smallest depth such that $b_i \le 8$ holds for all i and aims to minimize the variance of set $(b_0, \dots, b_{d-1})$. In such cases, 7+7+6 would be preferred over 8+8+4 or 10+10.

6 Experiments

6.1 Setup

We now examine performance of Vortex, whose code is available online [52], in various use cases from Section 2. This includes producer-consumer pipelines (both I/O and inter-thread), in-memory partitioning of keys, writing into an expanding array, and in-place sorting. Our benchmarks use Visual Studio 2019 on Windows and gcc 7.4.0 on Linux, both set to the maximum optimization level. We employ the hardware configurations in Table 1, each running an Intel i7 desktop CPU. The file system on $c_1 - c_2$ consists of 24 spinning hard drives (2 TB Hitachi Deskstar 7K3000), organized into RAID-5 and driven by two Areca 1880ix controllers. On the other hand, c_3 runs an NVMe SSD (Samsung 960 Evo) over the M.2 interface on the motherboard with a direct PCIe 3.0 x4 (4 GB/s) link to the CPU.

6.2 OS Bottlenecks

We start by benchmarking the rate at which Linux and Windows can slide a single 1-MB block along a virtual buffer using Vortex-B/C algorithms. Unless mentioned otherwise,

Table 2. Sliding-Block Speed on c_3 (GB/s)

Test	Linux		Windows	
	w/o faults	w/faults	w/o faults	w/faults
Vortex-B	5.9	5.9	5.7	5.7
Vortex-C	275	228	150	128

all experiments below use standard 4-KB pages. For Vortex-B, we commit one block, touch each of its pages, and decommit it. This process repeats sequentially for every block of the virtual space. For Vortex-C, we remap the previous block forward instead of returning it to the OS. To assess the penalty of catching page faults in user space, we consider both explicit block movement (i.e., the user performs commit/decommit/remap) and transparent (i.e., the fault handler does it instead).

The result is shown in Table 2, where a single thread can move blocks forward at over 220 GB/s on Linux and 125 GB/s on Windows. As expected, remapping in Vortex-C is significantly (i.e., $26 - 46\times$) faster than obtaining pages from the kernel in Vortex-B. It is also $13 - 23\times$ faster than `memcpy`, which maxes out at 12 GB/s on this host. Even with a user-driven fault handler, which reduces the speed by roughly 15%, both kernels easily exceed 100 GB/s. This is still immense compared to the speed of most producer/consumer pairs. While both operating systems post similar numbers in the first row of Table 2, Linux is almost twice as fast in the remapping test. Thus, in the remainder of this section, we only focus on Windows, whose results should be viewed as a lower bound on Vortex performance.

6.3 File I/O

Our first application is an I/O wrapper, built using overlapped and unbuffered calls into the kernel, which we compare against other known APIs. We use a 128-GB file (i.e., 4× RAM), where the write test produces 64-bit values into the file and the read test sums them up using SSE intrinsics.

Table 3 shows the result. Since c_1 and c_2 are identical in this test, we only discuss the former. We start with a C++ wrapper class `fstream`, which offers a stream-like interface to files. Its iterator provides users with one full token of data (e.g., a string or an integer), keeping buffer management hidden from the programmer. While `fstream`'s dismal speed (i.e., $50-70\times$ less than optimal) is not competitive against the proposed methods, this result illustrates a common situation with stream libraries and various high-level wrappers.

Usage of memory-mapped files in Windows is shown in the second row of Table 3. Server 2008 R2 on c_1 does only slightly better – 69 MB/s in the inbound direction and 147 MB/s outbound. In addition, it occupies the entire 32 GB of RAM with the file cache and ends up swapping idle applications to the pagefile. Server 2016 on c_3 is more peculiar. The read test completes fine @ 1,161 MB/s; however, the write benchmark freezes the machine for 7 hours. Technically, this

Table 3. File I/O Speed (MB/s)

Framework	c_1		c_3		CPU	RAM
	read	write	read	write		
`std::fstream`	43	88	51	140	8%	2 MB
Windows `MapViewOfFile`	69	147	1,161	*	8%	32 GB
Linux `mmap`	1,892	1,170	1,917	641	3%	30 GB
Vortex-A	2,235	1,547	1,272	651	8%	5 MB
Vortex-B	2,231	2,394	3,211	650	8%	5 MB
Vortex-C	2,238	2,399	3,266	674	1%	5 MB

Table 4. Batched Producer-Consumer Rate (GB/s)

Framework	Two cores			All cores			RAM
	c_1	c_2	c_3	c_1	c_2	c_3	
Storm [8]	1.7	1.4	2.4	11.1	9.5	12.8	1.6 GB
Naiad [41]	2.7	3.1	4.4	7.4	7.9	13.1	65 MB
Queue of blocks	6.4	7.3	11.4	17.1	16.5	24.8	24 MB
Vortex-B	4.3	4.4	4.6	5.1	5.2	3.9	9 MB
Vortex-C	13.5	16.4	23.3	38.3	38.4	65.4	9 MB

translates into 5.5 MB/s average rate, but because the OS is unusable all this time, we mark the result with an asterisk.

Linux `mmap` in the third row does significantly better. For example, it can read 25× faster than Windows on c_1 and 1.7× on c_3. However, its write speed on c_1 and read speed on c_3 are about half of the achievable rate, even when using `madvise` to declare sequential access to the file. This slow operation of `mmap` during file I/O is well-known, with more details and analysis provided in [16]. While their results suggest that opening the file with `MAP_POPULATE` can speed up data access, our tests indicate that this flag significantly increases the initial delay of `mmap`, resulting in worse overall runtime.

The next three rows show Vortex streams using $B = 1$ MB blocks. For read tests, we use $N = 1$, $L = 4$ (four overlapped read calls), and $M = 0$. For write tests, we reverse the roles of producer/consumer comeback, i.e., $L = 0$ and $M = 4$. As predicted, Vortex-A struggles in certain cases because it lacks multi-buffering. This becomes obvious in writing on c_1 and reading on c_3. Vortex-B fixes most of these problems, but consumes 8% of the CPU, which is roughly one full core occupied by the kernel memory manager. Vortex-C improves this further by lowering the CPU utilization to 1%. In all three cases, Vortex maintains just 5 MB of RAM.

6.4 Producer-Consumer

Our second application is an inter-thread data pipeline, where the producer generates 8-byte numbers and the consumer sums them up. Among the existing implementations, we examine Apache Storm [8], a prominent streaming system, and Naiad [41], a high-performance dataflow framework. For Storm, our producer batches up data in chunks of 1 MB, which we found to be optimal, before sending them for processing; pushing one key at a time yields performance 3 – 4 orders of magnitude lower. Since Naiad can be instructed to internally batch up the items, its operation is simpler as no special processing is required from the user.

To take advantage of SSE instructions, we also implement a version of this loop using a C++ iterator class, which internally allocates N blocks of size B per stream and sends pointers to them through a shared queue (see Algorithms 3-4). Experiments show that the best throughput is achieved using $N = 2$ and $B = 2$ MB. For maximum speed, each block is processed using vectorized intrinsics (i.e., `_mm_store_si128`, `_mm_load_si128`, `_mm_add_epi64`) and all functions from the iterator class are inlined. The Vortex user executes Algorithm 5, similarly modified to issue SSE instructions and run with the same values of (N, B).

In the first row of Table 4, Storm's single stream reaches 2.4 GB/s on c_3 and its rate across all cores peaks at 12.8 GB/s. Its main drawback is high RAM usage, i.e., 25× more than any of the other methods, which comes from the large number of packages it needs to load (i.e., ZooKeeper, Maven, Nimbus) and general inefficiencies of the Java runtime. Naiad performs better with a single stream, pushing over 4 GB/s on c_3, but cannot go much faster than Storm when utilizing all cores. Our version with a shared queue of block pointers doubles this performance, culminating in 24.8 GB/s.

Vortex-B, with one stream in the system (i.e., two cores), is capped to 4.6 GB/s. This is sufficient for some I/O devices, but is far too slow for data transport between threads, 100 Gbps networks, and SSD-based RAID setups. Going to multiple independent streams within a single process, the aggregate speed of Vortex-B does not increase much and, in one case, even goes down. This is caused by the Windows kernel not being able to parallelize commit/decommit requests.

On the other hand, Vortex-C is significantly faster. It more than doubles the rate of the block queue and exceeds the throughput of Storm/Naiad by 3 – 10×. To appreciate the numbers in the last row of the table, consider the Skylake-X architecture of c_3. A multi-threaded `memcpy` maxes out at 36.5 GB/s (i.e., 73 GB/s combined read/write bandwidth). In the same configuration, Vortex-C reaches 65.4 GB/s (i.e., 130.8 GB/s combined) using three concurrent streams, which is seemingly impossible. As it turns out, the consumer in Vortex-C reads from the L3 cache, which allows the system to run at almost double the speed of `memcpy`.

To follow up on the discussion in Section 2.2, we now revisit the issue of why iterators can be much slower (e.g., 2 – 3× in Table 4) than native access to pointers in memory-intensive benchmarks. Table 5 shows Intel performance counters from processing a stream with 100×2^{30} bytes, where we compare the queue of blocks (i.e., iterator) against Vortex-C. The column marked IR shows the number of instructions retired, which is followed by the number of loads/stores issued by the CPU. The last column records the number of cycles spent on each 16-byte `__m128i` item. The bottom row of the table presents the ideal Vortex values that would be

Table 5. CPU Performance Counters on c_3 (100-GB Stream)

	Producer			Consumer			Cycles
	IR	Loads	Stores	IR	Loads	Stores	per iter
Queue of blocks	87.3B	26.9B	13.4B	94.3B	20.2B	6.8B	6.6
Vortex-C	33.5B	1.7B	7.6B	51.6B	10.9B	2.2B	3.2
Ideal in theory	26.8B	0	6.7B	33.5B	6.7B	0	2.3

possible if a) the OS did not incur any overhead during page faults/remapping and b) inter-thread throughput were equal to the single-core bandwidth of the L3 cache (i.e., 33 GB/s).

Since the loop runs $100 \times 2^{30}/16 = 6.71$B times, the first row shows that the iterator producer issues 13 CPU instructions per item (in a loop with 38 instructions and two conditional jumps), four loads, and two stores. Out of these, only four instructions and one store, which are given in the last row of the table, would be needed if the program were written in assembly and no block boundaries existed. For the iterator consumer, the program executes 14 instructions per item (in a loop with 52 instructions and two conditional jumps), three loads, and one store, while the corresponding optimal values would be five, one, and zero. In the end, the iterator requires 6.6 cycles per item, which is 4.3 cycles higher than would be theoretically possible without block boundaries. Furthermore, had the stores been to L1 instead of L3, the iterator could have been 5.3× slower than the ideal rate (i.e., 1 cycle/item). Our analysis in Section 2.2 explains the issues faced by the compiler that generate this outcome.

For Vortex-C, the program runs with the optimal four instructions in the producer and five in the consumer, with no unnecessary memory traffic. Thus, the only difference between the second and third rows of Table 5 comes from the OS overhead. Because the consumer is generally faster than the producer, our implementation is asymmetric in the sense that the producer issues one map call, while the consumer does the rest of the work in Figure 5 (i.e., two unmaps and one map). Producer counters in Table 5 indicate that mapping 100 GB of space costs 6.7B instructions, 1.7B loads, and 0.9B stores in the kernel, most of which are probably wasted on spin-locks. The consumer side is predictably more expensive – 18B instructions, 3.2B loads, and 2.2B stores. In the end, we can estimate that the OS penalty of Vortex-C amounts to roughly 0.9 cycles for each 16-byte item, which is 4.8× less than the cost of the iterator.

6.5 Vanishing Array

Our next application is a *vanishing array*, which is a Vortex-S stream that has been simplified to omit handling of tricky cases from Figure 6. On startup, the array contains no data; as the user writes into the buffer pointer, the exception handler maps free blocks from a shared queue to the corresponding virtual addresses. When the user goes to read the buffer, page faults cause just-processed memory to be unmapped and inserted back into the queue, which makes it available elsewhere in the program (e.g., in other Vortex-S streams). This abstraction is useful when memory needs to transparently migrate from one place to another. For example, we can use a vanishing array for input keys during bucket sort, which allows empty pages to be reused in the buckets and later returned into the same buffer when the sort is finished.

Table 6. Populating an 8 GB Vector on c_3 (GB/s)

Framework	Memory	
	untouched	pre-faulted
`std::vector`	0.7	–
RUMA rewired vector, 4KB pages	–	5.3
RUMA rewired vector, 2MB pages	–	14.3
Chained blocks	6.8	18.8
Vanishing array (Vortex-S)	25.1	25.1
Regular buffer	8.0	28.5

Limited functionality of the vanishing array can be achieved using RUMA rewired vectors [46], which is a recent framework that supports buffer resizing using virtual memory. From the usage standpoint, their approach is quite different since RUMA does not catch page faults in user space and requires the application to manually call `push_back` on each item. This is similar to iterator-based streaming discussed in Section 2.2. On the technical side [45], their solution also takes a different route – it does not use anonymous pages and backs the virtual memory with files, which are created using either shared memory (4-KB pages) or `hugetlbfs` (2-MB pages). While RUMA pre-faults all pages ahead of time, the initial buffer is unmapped and new VMAs (virtual-memory areas) are created for subsequent writes as the buffer grows.

For the next test, we produce 8 GB of data into a resizing buffer. To achieve maximum speed, we use 16-byte items (i.e., `__m128i`) with streaming (non-temporal) SSE instructions. For RUMA, the sole method running on Linux, we use the provided `push_back` function, which is modified to invoke `_mm_stream_si128` for memory access. The result is shown in Table 6. On 4-KB pages, RUMA reaches 5.3 GB/s, which is 7.5× faster than `std::vector`. It gets a 2.7× boost from huge pages (i.e., 14 GB/s) because the kernel can rewire them much quicker; however, the remaining methods in the table achieve similar or better rates with standard 4-KB pages.

The next row in Table 6 is our C++ iterator implementation of vectors using chained blocks, which is a technique that starts with a pool of 1-MB buffers and chain-links them as the producer runs of out space during calls to `push_back` [43], [46]. It performs 31% better than the second row of RUMA, but scatters the data in disjoint locations in RAM, which may be a disadvantage in certain cases (e.g., when a binary search needs to run over the vector). The vanishing array performs identical in both columns, because physical pages on Windows do not incur faults on first access, and exceeds the best speed of chained blocks by 33%. The reason is similar to the one highlighted in the previous section – block-based

Table 7. Partitioning Speed of 8 GB on c_3 (M keys/s)

Framework	WC	$b = 8$	$b = 9$
Alg. 9 with 2-pass		339	322
Alg. 9 with chained blocks		450	413
Alg. 9 with Vortex-S		492	445
Alg. 9 with preallocated buckets		509	464
Alg. 9 with 2-pass	+	364	344
Alg. 9 with chained blocks	+	461	449
Alg. 9 with Vortex-S	+	607	523
Alg. 9 with preallocated buckets	+	637	567

algorithms incur penalties in checking boundaries, offloading registers to RAM, and running bloated loops.

In the last row, we provide the speed at which a process can directly write into an 8 GB buffer, which indicates that the vanishing array realizes 25.1/28.5 = 88% of usable RAM bandwidth on this machine. In consumer tests, where the vanishing array cleans up memory on read faults, it achieves similar levels of performance (i.e., 89% of the peak). In contrast, RUMA rewired vectors yield only 19% with 4-KB pages and 50% with 2-MB. In the latter case, the majority of the slow-down comes from compiler's inability to optimize `push_back`, even though the function gets inlined by gcc. Linux performance counters show that RUMA expends 10 CPU instructions, three loads, and two stores for each item, which is in agreement with our analysis in Section 2.2. This leads to 5.3 cycles per iteration of the loop, compared to 2.6 for the regular buffer in the bottom row of Table 6. With huge pages, the kernel is responsible for a negligible fraction of the runtime, which means that this 50% performance loss comes solely from the iterator interface.

6.6 Partitioning

Our fourth application partitions integer keys into 2^b bins, which is an elementary operation underlying radix sort. We implement four flavors of this task, all of which use the same optimizations and involve Algorithm 9 in the core. The only difference lies in how they solve the issue of a-priori unknown bucket sizes. These methods can perform complete sorts, but, for the purposes of this test, are stopped after producing level-0 splits. The first technique uses a counting pass to decide the starting offsets of buckets within an auxiliary array of size n. The second approach sends keys into buckets that dynamically expand using a chained set of 1-MB blocks, which allows it to use a single pass. Vortex-S is our third alternative, which is the only method that reads input from a vanishing array; all others use slightly faster regular buffers. The final method allocates buckets to the exact size needed, which represents the absolute best-case performance, not achievable in practice unless the distribution of keys is known in advance.

To achieve maximum speed, we ensure that any memory into which the application writes has been memset before partitioning begins. The result for uniform 64-bit keys is

Table 8. In-place Sort Speed (M keys/s)

Citation, type	Year	8 GB of keys			24 GB of keys		
		c_1	c_2	c_3	c_1	c_2	c_3
[21], MSB radix	2011	17.4	17.9	22.0	20.7	21.7	25.8
[43], MSB radix	2014	19.2	22.5	26.4	18.4	21.2	26.0
[48], MSB radix	2017	24.1	25.2	31.5	23.9	25.9	31.6
[42], MSB radix	2019	16.8	19.3	26.2	25.3	29.7	38.6
[9], quicksort	2017	21.8	23.1	27.6	21.3	22.4	26.9
`std::sort`	2019	9.5	9.7	11.6	9.0	9.2	10.9
[24], Intel TBB	2019	8.5	8.6	10.7	8.0	8.2	10.1

Table 9. Vortex-S Speed (M keys/s)

B	8 GB				24 GB			
	c_1	c_2	c_3	ϵ	c_1	c_2	c_3	ϵ
2^{20}	71	84	127	1.6%	68	80	121	1.1%
2^{19}	70	82	126	1.6%	67	78	121	0.8%
2^{18}	68	79	123	1.2%	65	75	119	0.7%

shown in Table 7, where a plus in the WC column indicates that the method uses software write-combine with non-temporal SSE instructions. The chained-block implementation is 26 – 32% faster than the 2-pass approach, which is consistent with our discussion in Section 2.3. Vortex-S improves this further by 7 – 31% and comes within 5-8% of the optimal speed in the last row. Because partitioning inherently runs at a lower rate (i.e., 3-4 GB/s) and does more work compared to the algorithms in the last two subsections, the chained-block iterator has a smaller *relative* penalty; however, performance counters still show a non-negligible amount of extra activity (i.e., 55% more CPU instructions, 14% more loads, and 31% more stores) compared to Vortex.

6.7 Sorting

Our fifth application of Vortex is in-place integer sorting using Algorithm 9. While the benchmarks below involve uniform 64-bit keys, the same principles can be easily adapted to sort 32-bit numbers, key-value pairs, and non-uniform workloads. For prior methods, we port all code to Windows and feed input from a regular buffer. On the other hand, Vortex-S reads from a vanishing array and outputs the result back into it, which allows it to remain in-place.

Table 8 lists seven competitor implementations for *in-place* sorting. The bottom three are comparison-based, while the top four are some of the fastest in-place radix-sort algorithms in the public domain. Where appropriate and known, we label rows with LSB or MSB depending on whether they use the least-significant or most-significant bit first. For each column, we also mark the fastest result using gray shading. Table 9 displays performance of Vortex-S on the same input, where block size B ranges from 1 MB in the first row (optimal speed) to 256 KB in the last row. We additionally display the fraction ϵ of memory occupied by partially-filled blocks at the end of each bucket stream. These results demonstrate

Table 10. Out-of-Place Sort Speed (M keys/s)

Citation, type	Year	8 GB			24 GB		
		c_1	c_2	c_3	c_1	c_2	c_3
[44], LSB radix	2011	24.6	24.7	39.3	unable to run		
[43], LSB radix	2014	23.8	26.4	42.0			
[50], LSB radix	2016	19.2	23.3	34.0			
[30], MSB radix	2017	25.2	29.4	41.0			
[29], MSB radix	2017	43.6	58.2	67.0			
[23], Intel IPP	2019	9.3	9.8	42.3			

Table 11. Speedup Factor of Vortex-S

Compared to	8 GB			24 GB		
	c_1	c_2	c_3	c_1	c_2	c_3
Best in-place	2.9	3.3	4.0	2.7	2.7	3.1
Best out-of-place	1.6	1.4	1.9	unable to run		
`std::sort`	7.5	8.7	10.9	7.6	8.7	11.1

that Vortex-S achieves a substantial improvement over the previous methods, while incurring negligible RAM overhead ϵ. On Skylake-X (i.e., c_3), it beats the fastest in-place methods [42], [48] by $3-4\times$ and STL quicksort by $11\times$.

While Vortex-S is hands-down the fastest technique that can sort 24 GB of keys on these machines, it is interesting to see how its performance stacks up against the best *out-of-place* methods. Table 10 shows six additional implementations, all based on radix sort, and their speed on 8-GB arrays. As expected, out-of-place methods are noticeably faster than those in Table 8, where the speed advantage of the current champion RADULS2 [29] over the top in-place sort for 8 GB arrays [48] is $1.8-2.3\times$. However, the candidates in Table 10, including Intel's optimized library IPP, are still no match for Vortex-S. On Skylake-X, our implementation posts rates that are $1.9-3.7\times$ higher. Comparison between the best method in each category and Vortex-S is summarized in Table 11.

7 Related Work

Operating systems have used virtual memory and paging for decades [49]; however, explicit reliance on these features in *user space* is more rare. One closely related paper in recent literature is RUMA [46], which is a framework that offers a *rewired-vector* abstraction that allows dynamic expansion of arrays using virtual memory. While Vortex can achieve RUMA functionality as well, its capabilities and speed go far beyond those of rewired vectors. There are several important differences between the two platforms as we discuss next.

First, RUMA equips the user with a vector push-back interface, which acts as a write-only iterator over an internal buffer. To decide whether the underlying array should be expanded, the iterator must reload the current write pointer from RAM and perform boundary verification on *each memory access.* As discussed earlier, this results in unnecessary work being done in the tight loop of the application, which significantly reduces performance. On the other hand, Vortex avoids this overhead by presenting the compiler with an infinite-buffer abstraction, which ensures that the hot path contains no unnecessary CPU instructions.

Second, RUMA does not catch page faults in user space, which prevents its application in software that is not aware of the syntax needed to interface with this framework. In contrast, Vortex can work with any C program, including pre-compiled libraries, as long as they know how to use a `char*` pointer. Third, RUMA does not perform memory cleanup on reads, which is a crucial characteristic that allows Vortex to achieve high-performance streaming and in-place operation. Finally, RUMA cannot support concurrent production and consumption out of a buffer, which is one of the main scenarios of interest in large-scale data processing.

Our application of Vortex to sorting can be viewed as a superset of the methods proposed in [53]. Their technique uses an MSB radix split at level-0, after which it switches to LSB within each bucket. Unlike Vortex, which uses single-pass MSB for *all* levels of recursion and runs in-place, the method in [53] neither releases memory on page faults nor moves physical blocks between buckets, which makes it out-of-place. In addition, their technique grows buckets using slow commit requests of Vortex-B rather than physical-page remapping, requires a counting pass at least once during the sort, hardcodes the split factor at $b = 8$ regardless of input size, and has to perform 8 levels of partitioning on uniform 64-bit keys (instead of $\log_2(n)/b$ in our sort).

Other uses of virtual memory in user space include pointer swizzling during page faults in object-oriented databases [54], maintenance of sorted lists in MonetDB [13], and hybrid approaches where a modified kernel helps applications achieve some desirable functionality [22], [56]. These papers do not address streaming scenarios considered here and their techniques are orthogonal to ours.

8 Conclusion

The Vortex paradigm offers a simple and high-speed programming model for manipulating large volumes of sequentially presented data, enabling memory-sensitive applications that would be impossible otherwise (e.g., fast in-place radix sort). The proposed approach is highly flexible since new fault handlers can seamlessly co-exist with various already-created streams within the same process. This opens up avenues for the community to contribute to the development of Vortex and incorporate its interfaces into current, as well as future, applications, languages, and frameworks.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1717982. We are also grateful to Tim Harris for excellent suggestions towards improving the paper.

References

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlander, M. J. Sax, S. Schelter, M. Hoger, K. Tzoumas, and D. Warneke, "The Stratosphere platform for Big Data Analytics," *VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[2] Apache Apex. [Online]. Available: https://apex.apache.org/.

[3] Apache Flink. [Online]. Available: https://flink.apache.org/.

[4] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/.

[5] Apache Kafka. [Online]. Available: https://kafka.apache.org/.

[6] Apache Samza. [Online]. Available: https://samza.apache.org/.

[7] Apache Spark. [Online]. Available: https://spark.apache.org/.

[8] Apache Storm. [Online]. Available: https://storm.apache.org/.

[9] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place Super Scalar Samplesort (IPS^{4}o)," in *Proc. European Symposium on Algorithms*, Sep. 2017, pp. 9:1–9:14.

[10] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, Main-memory Joins: Sort vs. Hash Revisited," *VLDB Endow.*, vol. 7, no. 1, pp. 85–96, Sep. 2013.

[11] O. Birkeland, "Searching Large Data Volumes with MISD Processing," Ph.D. dissertation, Norwegian University of Science and Technology, Sep. 2008.

[12] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A Comparison of Join Algorithms for Log Processing in MaPreduce," in *Proc. ACM SIGMOD*, Jun. 2010, pp. 975–986.

[13] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine," in *Proc. ACM SIGMOD*, Jun. 2006, pp. 479–490.

[14] A. Burnetas, D. Solow, and R. Agarwal, "An analysis and implementation of an efficient in-place bucket sort," *Acta Informatica*, vol. 34, no. 9, pp. 687–700, Sep. 1997.

[15] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, "PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort," *VLDB Endow.*, vol. 8, no. 12, pp. 1518–1529, Aug. 2015.

[16] J. Choi, J. Kim, and H. Han, "Efficient Memory Mapped File I/O for In-Memory File Systems," in *Proc. USENIX HotStorage*, Jul. 2017.

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.

[18] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm," in *Proc. VLDB*, Aug. 2002, pp. 299–310.

[19] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu, "Radix Sorting with No Extra Space," in *Proc. European Conference on Algorithms*, Oct. 2007, pp. 194–205.

[20] Google Cloud Dataflow, "A fully-managed cloud service and programming model for batch and streaming big data processing," 2016. [Online]. Available: https://cloud.google.com/dataflow/.

[21] E. Gorset, "In-place Radix Sort," Apr. 2011. [Online]. Available: https://github.com/gorset/radix.

[22] K. Harty and D. R. Cheriton, "Application-controlled physical memory using external page-cache management," in *Proc. ACM ASPLOS*, Oct. 1992, pp. 187–197.

[23] Intel Corporation, "Intel Integrated Performance Primitives," Mar. 2019. [Online]. Available: https://software.intel.com/en-us/intel-ipp.

[24] Intel Corporation, "Intel Threading Building Blocks," Mar. 2019. [Online]. Available: https://www.threadingbuildingblocks.org/.

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proc. ACM SIGOPS/EuroSys*, Mar. 2007, pp. 59–72.

[26] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Processing Engines," Feb. 2018. [Online]. Available: https://arxiv.org/abs/1802.08496.

[27] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs," *VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[28] D. Knuth, *The Art of Computer Programming, Vol. III*, 2nd ed. Addison-Wesley, 1998.

[29] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, "Even Faster Sorting of (Not Only) Integers," in *Proc. International Conference on Man-Machine Interactions*, Oct. 2017, pp. 481–491.

[30] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, "Sorting data on ultra-large scale with RADULS," in *Proc. International Conference: Beyond Databases, Architectures and Structures*, Sep. 2017, pp. 235–245.

[31] J. Kreps, "Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)," Apr. 2014. [Online]. Available: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[32] Kriss and other contributors. Fastest sort of fixed length 6 int array. [Online]. Available: https://stackoverflow.com/questions/2786899/fastest-sort-of-fixed-length-6-int-array.

[33] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proc. USENIX OSDI*, 2012, pp. 31–46.

[34] H. Liu and H. H. Huang, "Graphene: Fine-Grained IO Management for Graph Computing," in *Proc. USENIX FAST*, Feb. 2017, pp. 285–299.

[35] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A Performance Comparison of Open-Source Stream Processing Platforms," in *Proc. IEEE Globecom*, Dec. 2016.

[36] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *Proc. VLDB*, Aug. 2012, pp. 716–727.

[37] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "MOSAIC: Processing a Trillion-Edge Graph on a Single Machine," in *Proc. ACM EuroSys*, Apr. 2017, pp. 527–543.

[38] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. ACM SIGMOD*, Jun. 2010, pp. 135–145.

[39] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at What Cost?" in *Proc. USENIX HOTOS*, 2015, pp. 1–6.

[40] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber, "Cache-Efficient Aggregation: Hashing Is Sorting," in *Proc. ACM SIGMOD*, May 2015, pp. 1123–1136.

[41] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proc. ACM SOSP*, Nov. 2013, pp. 439–455.

[42] O. Obeya, E. Kahssay, E. Fan, and J. Shun, "Theoretically-Efficient and Practical Parallel In-Place Radix Sorting," in *Proc. ACM SPAA*, Jun. 2019, pp. 213–224.

[43] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort," in *Proc. ACM SIGMOD*, 2014, pp. 755–766.

[44] A. Reinald, P. Harris, R. Rohrer, and J. Dirk, "Radix Sort," 2011. [Online]. Available: http://www.cubic.org/docs/download/radix_ar_2011.cpp.

[45] RUMA Source Code. [Online]. Available: https://bigdata.uni-saarland.de/publications/rewiring_codebase.zip.

[46] F. M. Schuhknecht, J. Dittrich, and A. Sharma, "RUMA has it: rewired user-space memory access is possible!" *VLDB Endow.*, vol. 9, no. 10, pp. 768–779, Jun. 2016.

[47] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, "On the Surprising Difficulty of Simple Things: The Case of Radix Partitioning," *VLDB Endow.*, vol. 8, no. 9, pp. 934–937, May 2015.

[48] M. Skarupke, "Ska Sort," May 2017. [Online]. Available: https://github.com/skarupke/ska_sort.

[49] W. Stallings, *Operating Systems: Internals and Design Priniciples*, 8th ed. Prentice Hall, 2014.

[50] S. Thiel, G. Butler, and L. Thiel, "Improving GraphChi for Large Graph Processing: Fast Radix Sort in Pre-Processing," in *Proc. ACM IDEAS*, Jul. 2016, pp. 135–141.
[51] R. Vernica, M. J. Carey, and C. Li, "Efficient Parallel Set-similarity Joins Using MapReduce," in *Proc. ACM SIGMOD*, Jun. 2010, pp. 495–506.
[52] Vortex. [Online]. Available: http://irl.cs.tamu.edu/projects/streams/.
[53] J. Wassenberg and P. Sanders, "Engineering a Multi-core Radix Sort," in *Proc. Euro-Par*, Aug. 2011, pp. 160–169.
[54] S. White and D. DeWitt, "QuickStore: A High Performance Mapped Object Store," in *Proc. ACM SIGMOD*, May 1994, pp. 395–406.
[55] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters," in *Proc. ACM SIGMOD*, Jun. 2007, pp. 1029–1040.
[56] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "CRAMM: Virtual Memory Support for Garbage-collected Applications," in *Proc. ACM OSDI*, Nov. 2006, pp. 103–116.