A buffer management system (BMS) governs all communication between filters and applications. Its design determines the number of copies, data-cache hits and task switches per record. This chapter presents a streaming I/O BMS that minimizes cost by building streams from shared memory regions (4.2) and replacing copying with pointer indirection (4.3). Buffer specialization (4.4 and 4.5) ensures high throughput for varied applications and systems.
4.1 Overview

To maximize throughput, technical buffering concerns must be hidden from users and tailored by the control system. The BMS in Streamline \(^1\) presents applications and filters only with an idealized view of streams as sliding windows over unbounded, ordered sequences of blocks. All streams are represented as Unix pipes, because that interface fits the task and is well known. Underneath, three optimizations minimize copy-, context switch- and cache miss overhead.

**Shared Rings (4.2)** replace per space private data copies with shared access. To allow asynchronous access by multiple filters in multiple spaces, blocks are stored in large circular buffers and on a shared memory system these are mapped between spaces. To guard against corruption, shared buffers are immutable. Shared rings *minimize copying*. Fewer copies translates into a smaller working set, smaller cache footprint and therefore *fewer cache misses*. The lock-free nature of ring buffers *reduces mandatory context-switching*.

**Indirection (4.3)** replaces direct data manipulation with pointer arithmetic. Pointer queues replace the data queues in a store and forward I/O network with pointer queues that point into shared data rings. This model *avoids copying* between filters. Because they are private, pointer queues are mutable. Pointers can be manipulated instead of data to perform data reordering and copy-on-write. In Streamline, indirection is independent from memory addressing to operate seamlessly across virtual memory protection domains and physical device regions.

**Specialization (4.4)** of buffers matches their implementations to stream features, such as data pattern and synchronization strategy. Matching buffers to data pattern avoids fragmentation, *eliminating data cache misses*. A switch to permissive synchronization strategies *reduces context switching*. Section 4.5 introduces specific specializations.

4.2 Shared Rings

A straightforward BMS alternates filters with queues, implementing a store and forward network. Storing and retrieving data at each hop is not an efficient strategy, however, when data access is a significant cause of overhead.

---

\(^1\)Presented individually as Beltway Buffers [dBB08a]
The previous chapter showed how signaling between filters can be reduced; this section presents a solution to avoid data copying.

**Shared Buffers** Traditionally, the pipe interface is implemented using system calls, but Streamline streams are based on local function calls to make the interface available *everywhere* in the system. Function calls avoid the context switch and TLB shootdown overhead caused by system calling by operating on local memory buffers. To avoid copying, buffers are mapped into all interested memory protection domains. Previous work has shown that modifying virtual memory mapping in this manner is cheaper than copying [PAM94]. The BMS increases these savings by avoiding per-block virtual memory operations, instead reusing the same mappings for whole streams. It scales buffers to encapsulate not just single blocks, but whole streams. We will demonstrate that the reduction in mode transitions increases small block throughput and that the reduction in copying does the same for large blocks.

**Ring Buffers** Operating systems commonly allocate blocks on-demand and order these into streams using pointer queues. In contrast, the BMS stores blocks sequentially in a shared buffer to create a shared circular data buffer, or *DBuf*. Shared ring buffers have previously been shown to reduce transport cost between the kernel and userspace processes [GA91, BdBC+04]. Long-lived shared rings hold six implementational advantages over dynamically allocated structures:

1. They amortize allocation cost over many blocks.

2. They amortize virtual memory (mapping) operations in the same manner.

3. They are lock-free.

4. They require fewer memory accesses than pointer structures.

5. They minimize cache collisions by ordering data sequentially.

6. They assist hardware prefetch for the same reason.

### 4.2.1 Networks of Rings

We observe four obstacles to moving to a buffer architecture based on a coarse-grain ring, all of which are resolved by moving to networks of rings.
Memory Waste  Rings trade memory utilization for speed. As memory
density grows faster than bandwidth, trading off space for speed is increas-
ingly appropriate. Overprovisioning can be curtailed, however, by selecting
buffers optimized to stream and system features. The BMS can specialize
buffer size and layout.

Process Isolation  In a multitasking OS, processes must be isolated from
one another to assure correctness and privacy. A naive implementation of
shared rings breaks this isolation property. Access control is more difficult to
enforce when rings are coarse-grain structures, since per-block access poli-
cies cannot be enforced. With only a few policy groups and devices (the com-
mon case), security issues can be resolved by switching to a multi-ring ar-
chitecture. Fine-grained protection domains is reestablished by having mul-
tiple rings with private access control policies. Each Streamline buffer car-
rries a policy that follows familiar Unix file permissions. Access is set for the
user, group and globally, for both reading and writing. Data manipulation
of filters with only read-only access to a buffer is automatically reflected to a
secondary buffer that has the right policy. Protection is enforced per ‘buffer
plus computation space’ pair, because the BMS can check permissions only
once, before mapping in the I/O region. As a consequence, policy enforce-
ment causes no other run-time overhead.

Cache Invalidation  Multiprocessing can cause pathological cache be-
havior, whereby one CPU causes invalidation of lines in another’s cache with-
out any real data dependency. Such false data dependencies must be avoided
to maintain consistent high efficiency. This BMS splits buffer metadata into
shared read-only and private writable regions, similar to netchannels [JF06].

NUMA Access  Shared memory is often, but not always the preferred trans-
fer mode. On NUMA architectures, explicit copying between buffers is faster
than shared memory access, because it reduces contention on the (high-latency)
interconnect. For instance, some network cards, like the Radisys ENP2611,
have large amounts of memory into which packets are received and that can
be shared with the host processor. It is cheaper to DMA the network data
across the PCI bus once, than it is to have the host processor read data from
a shared buffer on the card [NCdB04]. The BMS transparently interposes
a write-through cache between high latency clients and data, that appears
to give direct read access to shared contents, but underneath replicates data
into a local DBuf. Replication policies are specialized to hardware features, to
exploit optimizations such as prefetching, bursting or direct cache access.
The BMS exploits the demand for multiple buffers by tailoring each buffer to application and system. For example, device drivers adapt buffers to fit device hardware specification, while host network packet rings align containers to the start of the network layer protocol header. To avoid cross-buffer copying, the BMS weaves the set of buffers into an integrated network of rings through indirect buffers.

4.3 Indirection

Presenting clients with an idealized view of private, sequential streams conflicts with copy-avoidance through shared buffers. Modifying shared blocks can cause data corruption, for instance. Copying blocks between buffers is a safe alternative, but that is expensive. Instead, Streamline takes an approach similar to hardware protected virtual memory. It adds a layer of indirection in memory addressing, to be able to virtually move and copy blocks without incurring any actual physical data movement. The BMS implements indirection in software, because hardware-protected virtual memory operations are expensive, protection is not required by Streamline (where access control takes place at buffer granularity), and software indirection can be enabled selectively to curtail translation cost.

Virtual address pointers are meaningless across memory protection domains. In the BMS, software indirection replaces pointers with globally valid indices and pointer queues with index buffers or “IBufs”. Indices are pointers that differ in two ways from memory addresses. One, they replace direct addressing with a hierarchical lookup structure that is valid across virtual address spaces, a rich index. Two, their structure embeds a metadata field. Figure 4.2 shows an example index. It strongly resembles the tag used in stream classification. IBufs are specialized ring buffers that pack indices closely together to share cachelines, maximizing cache hitrate. Figure 4.1 shows mul-
multiple (private) IBufs pointing into (shared) DBufs.

**Rich Pointers** Since data is shared globally, indices must address buffer contents across memory protection regions. A rich index is a three-level lookup structure consisting of a globally unique DBuf identifier, a block offset within this buffer and an offset plus length pair to select a region within the block (e.g., a TCP segment within an Ethernet frame). Indices from different IBufs may share access to the same DBuf and indices within the same IBuf may point to blocks in multiple DBufs. The first situation is common when multiple clients need a private view on data residing in a shared ring, for instance the network packet reception ring. The second situation occurs when a client needs to access multiple rings, e.g., when a server listens on two NICs at once, each of which controls its own descriptor ring. Figure 4.1 shows both situations.

Resolving rich pointers is more expensive than following virtual memory pointers due to lack of hardware address translation. The translation cost is amortized by caching a translation for subsequent accesses within the same space. More expensive is the mapping in of previously unmapped buffers – what we call “buffer-faults” in reference to hardware page faults. Such exceptions occur when a DBuf is referenced that is not mapped into the current memory protection domain. To maintain the illusion of globally shared buffers, such faults are handled automatically, similar to demand paging. If the space has the appropriate access permissions the pages underlying the buffer are mapped into the address space all at once ².

---

²The current implementation is based on an OS memory management interface and unnecessarily suffers a hard page fault for each individual page.
Transparent Indirection  

Like virtual memory, indirection is transparent to users. Because index and buffer-fault resolution are non-trivial operations, the BMS automates these steps behind the buffer interface. Recall that all indirection takes place in IBufs, which are specializations of the basic DBuf ring. The IBuf interface is identical the DBuf interface, rendering implementation distinctions immaterial to applications. The IBuf implementation resembles a cache. Reading from an IBuf is a two step process of rich pointer resolution followed by a read-through into the referenced DBuf. Similarly, writing to an IBuf is implemented as write-through to a DBuf followed by a store of an index into the IBuf. Users do not explicitly specify the DBuf for the write-through operation; a default buffer with the same access control policy as the IBuf is automatically selected. If none exists, one is created. Note that write-through is rare in practice, as a block referenced in an index is generally already stored in a DBuf.

4.3.1 Splicing

If write-through is avoided, a virtual data copy appears without a physical copy operation having taken place. Such copy-free data transport is known as splicing [McV98]. It is more efficient than copying for all but the smallest blocks. One logical application of splicing is network transmission queuing. Application payload is invariably already stored (in application buffers or the file cache), so there is no need to create a separate copy for transmission. In an IBuf write, Streamline compares the passed source pointer to the address ranges of up to N DBufs, whereby we choose N so that the list of ranges fits in a single cacheline to minimize lookup cost. If a list entry matches, Streamline skips write-through, calculates a rich index and only writes this to the IBuf. Splicing has also been integrated into the Linux kernel with version 2.6.17. That implementation differs from ours in two important ways. First, it introduces a new interface independent from regular data transfer and is therefore not backwards compatible. Second, it only governs movement of data within the kernel. While it offers the same performance advantages as Streamline in certain situations, few applications have so far been adapted to the Linux splicing model. Splicing in Streamline, on the other hand, is backwards compatible with existing applications. Again, buffers present the Posix I/O interface; applications are unaware of which transfer method is used underneath.

A second optimization increases throughput for the common operation of reading a block from one stream and writing it unmodified to another. As presented, write splicing only reduces the cost of the write call, application splicing avoids the copy into the application buffer during read. The technique is based on lazy copying. In this call, the system temporarily revokes
access to the page(s) underlying the buffer. If the application forwards data unmodified, write will receive the same application buffer pointer as read. The call then splices from the originating DBuf. Again, Streamline caches the last N pointers for lookup. If a page-fault occurs, instead of splicing, data is copied lazily. Splicing is then disabled for that DBuf, as frequent page-fault handling actually degrades performance.

**Results** We now quantify the effects of write splicing on throughput by comparing a regular read/write cycle with one that replaces the read with a direct pointer from the input buffer. This pointer is obtained with a call to peek, a read-like function that we will explain shortly. Figure 4.4 shows the performance of moving data from a DBuf to an IBuf. The test is indicative of file servers, for instance, where data is read from the page cache and written to the network transmission buffer. The fastest mechanism is peek only: the peek equivalent of read-only access. This mechanism processes even faster than the physical bus permits, because no data is touched. The method serves no purpose; we only show it to set an upper bound on the performance. About half as fast is fast peek/write, which combines peek with splicing. This, too, does not actually touch any data, but writes out an IBuf element. Overhead caused by read can be seen by comparing these two results with those of read only and fast read/write. They are 3x slower still. Worst results are obtained when we cannot use splicing, but instead must write out data: throughput drops again, by another factor 2.5. This experiment clearly shows that combined gains from copy avoidance are almost an order of magnitude (9x) when all data is cached. Savings will be even higher for buffers
that exceed L2, because then the large blocks will cause many more DCache and TLB misses than the small IBuf elements.

4.4 Specialization

To efficiently handle wide ranges of tasks, buffer features are specialized. Centralizing the logic to handle all cases imposes an unacceptable high cost on simple common use of pipes. Examples of useful specializations are cache-aligned storage for small records such as IBufs and resizable buffers for streams that fluctuate widely in throughput. This section shows the uniform features that all buffers must share: a common interchangeable interface support for specialization and a small set of shared implementational features.

Implementation Stacking  Because an exhaustive list of optimizations cannot be compiled, the BMS does not export specialization hooks for, say, synchronization. Instead, buffer implementations specialize arbitrary behavior by implementing a complete stream interface. All buffers implement a smaller, simpler, but generally similar interface to the external Unix I/O API. This form of specialization is very similar to virtual filesystem implementations, as for instance found in Linux. Like a VFS, the core BMS logic only resolves a desired implementation and performs slow control tasks, such as

---

Figure 4.4: Write Splicing
namespace management and segment mapping. Some features are orthogonal to one another and can in principle be combined at will. For example, synchronization and data layout concerns are completely independent. When all implementations export the same API, calls can be performed iteratively, or stacked, to combine features (at the cost of extra function calls).

4.4.1 Interface

To be able to replace implementations at will for non-functional reasons, all buffer implementations must expose the same interface. In Streamline, buffers present the Unix I/O interface, because it is familiar and convenient (as the basis for Unix pipes). To differentiate between the system calls made available by the OS and the BMS functions of Streamline the latter an $\text{slfile}_e$-prefix. The BMS presents Unix calls to applications. Internally, it translates these calls into the more fine-grained API presented to implementations. Applications can use both the coarse-grain legacy and fine-grained extended interface. Table 4.1 summarizes the combined API.

**Extending the Unix Interface**  One performance drawback of Unix I/O is that it implements expensive copy semantics, that is, read and write create private copies of blocks for the caller. These semantics are safe, but also wasteful, as they must be implemented using copying or VMM modifications such as copy-on-write. To circumvent these costs we extend the API with read-through and asynchronous write calls. Peek is a read-like operation that uses weak move semantics in the nomenclature of Brustoloni and Steenkiste [BS96]. We prefer to call the scheme system shared, because no action, move or otherwise, is implied. With system shared semantics, a caller is not allowed to modify data, because it is given a direct pointer into the DBuf. This behavior can only be enforced at a whole-ring level. The function is identical to Unix read apart from the second argument, which takes a double instead of a single pointer:

```
ssize_t slfile_peek(int fd, void **buf, size_t count);
```

When peeking, a client receives a direct pointer into the stream. The read call, then, is nothing more than a wrapper around a peek call, memcpy and exception handling logic. Figure 4.5 shows the gains obtained by switching from copy based reading (R) to an indirect peek (P) call for multiple buffer sizes. The figure plots read throughput using both methods at various call sizes. Upper and lower quartiles are within 4% of the presented results. As expected, peek throughput scales linearly for all buffers, as it is purely computational. Read throughput, on the other hand, experiences memcpy overhead. Even for the smallest packets, it is about one third slower than peek. Where
<table>
<thead>
<tr>
<th>Function call</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read access</strong></td>
<td></td>
</tr>
<tr>
<td>slfile_read</td>
<td>access data at the read pointer through a private copy</td>
</tr>
<tr>
<td>slfile_peek</td>
<td>access data at the read pointer read-through</td>
</tr>
<tr>
<td>slfile_peek_acquire</td>
<td>like above, but do not update the read pointer</td>
</tr>
<tr>
<td>slfile_peek_complete</td>
<td>update the read pointer</td>
</tr>
<tr>
<td><strong>Write access</strong></td>
<td></td>
</tr>
<tr>
<td>slfile_write</td>
<td>add data at the write pointer from a private copy</td>
</tr>
<tr>
<td>slfile_poke</td>
<td>add data at the pointer write-through</td>
</tr>
<tr>
<td>slfile_poke_acquire</td>
<td>like above, but do not update the write pointer</td>
</tr>
<tr>
<td>slfile_poke_complete</td>
<td>update the write pointer</td>
</tr>
<tr>
<td>slfile_append</td>
<td>in a record-oriented ring, append to the open record</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td>slfile_check</td>
<td>generate timestamps and verify data validity</td>
</tr>
<tr>
<td>slfile_block</td>
<td>block on a single stream</td>
</tr>
<tr>
<td>slfile_select</td>
<td>block on a group of streams</td>
</tr>
<tr>
<td>slfile_lseek</td>
<td>seek in the stream</td>
</tr>
<tr>
<td>slfile_open</td>
<td>open a stream</td>
</tr>
<tr>
<td>slfile_close</td>
<td>close a stream</td>
</tr>
<tr>
<td><strong>IPC</strong></td>
<td></td>
</tr>
<tr>
<td>slfile_pipe</td>
<td>open a stream with Unix pipe semantics</td>
</tr>
<tr>
<td>slfile_mpipe</td>
<td>open a pipe with a user-defined number of readers</td>
</tr>
<tr>
<td>slfile_mkfifo</td>
<td>create and open a Unix named pipe</td>
</tr>
<tr>
<td>slfile_fork</td>
<td>create a new space that shares all open streams</td>
</tr>
</tbody>
</table>

Table 4.1: The extended stream interface

possible (internally and for fault-tolerant applications such as P2P clients), Streamline peeks instead of reads.

Peeking is dangerous with respect to synchronization: it increases the consumer index before the consumer has finished accessing data, which can cause a conflict with a fast producer. The BMS solves this problem in two ways: for optimistic processing, where conflicts are rare and non-destructive, no action at the block level may be necessary. This is true when application-level integrity checks are supplied (e.g., in BitTorrent clients) or when timeliness is more important than correctness (e.g., for streaming media). It may be helpful when overwrites can be detected after data handling, for instance to discard video frames. For this purpose, the function

```
i64_t slfile_check(int fd, i64_t timestamp, int flags)
```
generates and compares stream-private epoch counters. Under normal operation it returns, depending on the mode parameter, a counter related to the current read or write pointer. If an application wants to verify that data read, respectively written at or after a given check is valid, it calls the function again and passes the previously obtained counter. If the data indeed was valid, a new timestamp is returned that can verify the current state, otherwise an error code is given. The operation only involves a single memory lookup (of the shared read- or write counter) and is therefore very cheap. The dual function of retrieving and verifying timestamps enables continuous optimistic peeking in a tight, fast loop:

```c
    t = -1;
    while ((t = slfile_check(fd, t, O_RDONLY)) != -1) {
        peek_stream(fd, &ptr, SIZE_MAX);
        /* process data */
    }
```

Checking after use is an optimistic strategy. If data loss is not acceptable, peeking can instead be secured defensively by separating the data access and pointer increment operations. The resultant interface pair implements asynchronous read access and requires more extensive application changes than a change from `read` to `peek`.

```c
    ssize_t slfile.peek.acquire(int fd, void **buf, size_t count)
    ssize_t slfile.peek.complete(int fd, size_t count)
```

Peek removes the copy during reading. Poke applies the same performance optimization to writing. The interface is practically identical, with
a single replacement for `slfile_write` called `slfile_poke` and an asynchronous pair for defensive programming.

```c
ssize_t slfile_poke(int fd, void **buf, size_t count);
ssize_t slfile_poke_acquire(int fd, void **buf, size_t count)
ssize_t slfile_poke_complete(int fd, size_t count)
```

To simplify the common task of protocol header generation, the BMS also provides a sequential append operation. While the asynchronous interface requires manual pointer arithmetic between the acquire and complete calls, this sequential interface only uses a chain of safe write-like calls. Contrary to write, append does not move the write pointer, but only adjusts the offset plus length pair of the open record. It also does not wrap (in other words, it does not implement a queue within a queue). The interface is identical to that of write:

```c
ssize_t slfile_append(int fd, const void *data, size_t count)
```

**Pipes**  Rings are opened as files, but as PipesFS showed, are represented in the file namespace as pipes. With little effort, arbitrary IPC is sped up through shared memory, by reimplementing the `pipe` and `mkfifo` calls to create a Streamline ring. The change is transparent, implemented through library interpositioning, so that the transport optimizations can apply to all Unix IPC applications. Because shared buffers are mapped into virtual memory as a whole, shared memory pipes can be accessed with arbitrary heap operations. Therefore, they clearly do not offer the full isolation of Unix pipes.

**Multiway Pipes**  Streamline's ring buffers are especially well suited to group communication, since they aggressively share memory. Unix I/O lacks simple group communication primitives, although these have many uses, especially on multicore systems. For this purpose, Streamline adds a multiway pipe interface: a group communication channel that resembles the Unix pipe, but that multiplexes all input onto all output channels. Mpipes trivially map onto multiconsumer rings. The call

```c
int mpipe(int fds[], int num);
```

creates a single producer descriptor and `num – 1` consumer descriptors, each with a private file offset. All communication takes place through shared memory.

**A Clean Slate Alternative**

This thesis extends the Unix I/O interface with calls that operate on shared memory and that understand record-based semantics, among others. By doing so, we arguably muddle the waters. The original Unix I/O Interface [RT74]
is a model of simplicity and clarity. It has at its core only four easily understood operations (open, read, write, close) that are valid across resources ranging from disk storage to character devices and IPC pipes. A valid question is whether an equally clear API could be developed today, if unencumbered by legacy constraints.

First, we observe that maybe the interface isn’t as simple as it initially appears. A crucial element in the observation about Unix I/O, is the statement ‘at its core’. A whole ecosystem of extensions is required to build practical, efficient applications. In this sense the Streamline functions are not new. The four basic operations are supplemented with others for, among others, truncation, seeking and syncing to persistent media. Over the years, this supporting set has grown with methods for notification handling (epoll, SIGIO), asynchronous I/O (aio_write), locking, scatter-gather I/O (writev), buffered I/O (fstream) and even memory mapped access that voids all structured semantics (mmap). Even the existence of the simple create call indicates that Unix I/O is not as simple as is often believed – at least from the point of view of an implementer. Even after decades of use, no stable, canonical API has emerged. With its 2001 revision, the Posix standard introduces the aio family of asynchronous I/O functions and only in version 2.6 does Linux bring the splice interface (Section 4.3.1, a feature that we at least integrate behind the core calls). The main reason for extending Unix is that so much is built on the core interface that replacing everything is impractical. Instead, a developer is encouraged to speed up the few hot spot calls in his or her few application and to rely on the core API for the non-critical codepath. Over time, this development model adds layers of interface that become de facto standards themselves, expanding the tarpit that is legacy software.

A simple all-purpose interface, if feasible, should exhibit the main quality attributed to the core Unix calls: elegance through simplicity. In API design, as often elsewhere, perfection is achieved not when there is nothing more to add, but when there is nothing left to take away. In contrast, however, to the Unix calls, a new interface will have to deliver performance comparable to the known optimizations. Let us briefly consider the possible designs.

Zero-copy interfaces (splice/tee and poke/peek) offer the same interface sans copying, but in doing so weaken the contract between caller and callee over Unix, by allowing the caller to modify data after passing it to the callee. How serious a problem is this? For existing applications that bank on data being constant after the call, it clearly is not acceptable. But having taken leave of such legacy concerns, we say that this requirement is only infrequently im-

---

3Antoine de Saint-Exupery: “Il semble que la perfection soit atteinte non quand il n’y a plus rien à ajouter, mais quand il n’y a plus rien à retrancher.”
important. Moreover, when it is, it can be easily reinstated by performing an explicit copy behind the interface and calling check_stream before doing any other work. API design is an art, not a science, and in this case we feel that the infrequent need warrants offloading responsibility. Zero-copy (pointer-based) interfaces are a good idea.

Scatter-gather I/O (readv/writev) reduces call overhead by packing multiple blocks per call. These calls introduce structures that are unwieldy compared to simple pointers, but this is mainly an artifact of programming in C. A simple list-based interface is commonplace across the higher level languages that most programming today takes place in: not just in purely functional languages such as Lisp, but also the versatile Python and C-inspired C#.

Asynchronous I/O methods avoid blocking a thread of execution. Multithreaded programming is quickly becoming essential on multicore hardware, but we see no advantage in mixing a multithreading and a signal-handling model of computation in the same application. Control flow of the kind aio_read followed by aio_error followed by aio_return duplicates duplicates synchronization methods more generically implemented in thread or signaling libraries only for a specific application domain. For this reason we do not advocate this path.

If signal handling is preferred over threading, a single thread of execution should have only a single event handling loop. Niels Provost’s Libevent is a popular, because efficient, library built around this concept. Here, application code registers callback functions for given events. This programming model is remarkably similar to Streamline filter invocation. That goes one step further, however, and exploits the fact that filters do not perform explicit I/O to optimize data transfer automatically (Chapter 4). In fact, callback interfaces take away much control from the application. As a result, this is probably the most flexible of the presented efficient interfaces and for many applications to be preferred over explicit application-level I/O. Clearly, if we did not believe this, we would not have developed Streamline as a callback-based system.

Implicit I/O is not suitable for all situations, however. Streamline itself also exports the explicit Unix-like I/O interface for applications that want to inspect streams at any point in time (i.e., not when a specific event happens / thread unblocks) – which brings this discussion full circle. The simplest interface for I/O and synchronization is still the four-call Unix interface. If we were to design a new interface, we therefore would not select a completely different programming model, but instead would remove all extensions to these four calls that duplicate unnecessary behavior (e.g., creat) and would modify default parameters to use shared memory I/O and lists. The result is not perfect, but by removing the need for many other extensions it is arguably
4.4.2 Common structure

Implementations are specialized and extracted out of the common control flow as much as possible, but a non-trivial shared skeleton remains. At its simplest, a ring buffer consists only of a linear memory region and two indices: a read and a write pointer. In practice, extensions are needed. For example, to support implementation stacking, each buffer interface must maintain a runqueue of function pointers that it calls in-order for every external API call.

In multiuser rings, the metadata must be extended with a read pointer for each consumer. Access control enforcement must prevent read-only clients from writing to shared metadata, which requires splitting metadata into two types of segments: one for client private and one for shared protected metadata. Besides access control, separating the two indices has the advantage that it reduces cache thrashing resulting from concurrent updating to the same cacheline by producers and consumers. Usage counting and intra-space synchronization is handled through a per-space local metadata segment. Figure 4.6 shows the complete structure of a buffer, containing one data segment, one producer metadata segment, per-client private consumer metadata segments, per-client buffer implementation stacks and finally per-space local metadata segments. The local segment is straightforward, but the others embed shared state that warrants further attention.
**Shared Metadata Segment**  Data and producer metadata always have the same access policy. In principle, the two can be co-located in a single segment, but in practice, it is more convenient to separate them because performance often requires data segment size to be a power of two, while allocation of just over a power of two to include metadata is generally not supported for large contiguous regions. Besides the write pointer, producer metadata consists of access control and ownership, buffer characteristics such as size and runtime statistics such as droprate. Implementation-specific features add further metadata: event batching needs signals counters, multi-process access calls for locking.

**Private Metadata Segment**  The private metadata segment also embeds more items than a single index. It records usage statistics, such as throughput and droprate (in the case of lossy synchronization). More importantly, specialization requires a reference to the implementation specific API handlers. Implementations are not just tied to a buffer—in which case the shared producer segment could be used—but to a user, because specialization is also used to adapt to user-specific situations, such as high latency in the case of NUMA processing. To enable stacking of implementations, the segment maintains a array with pointers to buffer implementation structures.

**Addressing**  Block offsets can be given relative to the start of a linear memory region or absolutely from the start of the stream. We mention the trade-off explicitly, because the choice is consequential: some specializations require absolute indexing to function. In the first case, a ring is split into $N$ containers of one or more bytes and pointers run from 0 to $N$. Two data items that are stored in the same container at different occasions share the same pointer. Absolute pointers, on the other hand, uniquely identify each written element. Whereas container points wrap, so that offset $m \times N + b$ equals $b$ for all $m$, absolute pointers grow monotonically. When one item overwrites another they can both still be uniquely identified even if only one can be accessed. The Streamline BMS mandates absolute pointing. Five specializations in two sets currently use the added information. One set can vary buffer size, rendering relative offsets meaningless. A second can experience data loss. These specializations require absolute pointers to verify integrity. All specializations are discussed in detail in Section 4.5, where indexing is also discussed where pertinent.

**Locking**  Extensive metadata breaks the lock-free nature of rings. As long as increments to the two main indices can be guaranteed to be atomic,
## Dimension Options

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Static or dynamic</td>
</tr>
<tr>
<td>Allocation</td>
<td>Contiguous or pointer-based</td>
</tr>
<tr>
<td>Layout</td>
<td>Byte or record oriented</td>
</tr>
<tr>
<td></td>
<td>Compressed, aligned or neither</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Blocking or non blocking</td>
</tr>
<tr>
<td></td>
<td>Lossless or lossy</td>
</tr>
<tr>
<td></td>
<td>Throttled or Low-latency</td>
</tr>
<tr>
<td>Replication</td>
<td>Caching or read-through access</td>
</tr>
<tr>
<td></td>
<td>Burst copying, prefetching or random access</td>
</tr>
</tbody>
</table>

### Type Function

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirection</td>
<td>Transparently resolve IBuf contents</td>
</tr>
<tr>
<td>Filesystem</td>
<td>Filesystem namespace interface</td>
</tr>
</tbody>
</table>

Table 4.2: Specialized implementations in the reference system

Other locking is still not required on the fast path for generic buffers. Atomic increment can be implemented on most current CPUs. On simpler hardware or across devices, (bus) locks must also be acquired. The BMS does not currently select the cheapest atomic increment implementation automatically. With more than two clients, locks may also be necessary at runtime to support group communication or costly resize operations. Updates to metadata always require exclusive access. Because these are infrequent, optimizations such as reader-writer locks are not used.

### 4.5 Ring Types

Specializations can be grouped by the buffer characteristic they modify. We discern size, allocation method, data layout, synchronization and replication policy, but this list, summarized in Table 6.1, is not necessarily exhaustive. The second part of the table shows more fundamental, because functionally unique, specializations. IBufs modify read and write behavior to transparently access a DBuf. File wrappers incorporate persistent storage into the BMS. This section explains the use case and implementation of each specialization (except IBufs, which have already been discussed).
4.5.1 Size

As the memory system is the main bottleneck in streaming I/O [PS06], support mechanisms such as prefetching, burst transfer and caching must be exploited when available. Optimizing these features is difficult, because cache sizes and memory latencies differ widely. Besides specializing to optimize for specific hardware environments, we follow a more general rule-of-thumb: *a decrease in memory usage will lead to an increase in throughput*. Usage is not the same as allocation: we focus on minimizing the runtime memory footprint. Before we employ this rule we note its main exception: data alignment to hardware boundaries (cachelines, pages) benefits performance; squeezing the last bit out of each allocation is not our goal.

The size of a buffer affects its maximum throughput in two ways: larger buffers reduce synchronization overhead (such as task-switching), but smaller buffers experience fewer cache misses. In ring buffers, miss-rate increases abruptly when a buffer exceeds the size of a cache, because the common LRU (like) replacement policy will consistently evict the last accessed and thus first needed slot. On the other hand, task-switch cost (around 1000 cycles plus cache effects on modern CPUs) dwarfs cache miss overhead. Buffers must therefore fit in the smallest cache that does not introduce excessive task-switching. For Ethernet frame-sized blocks, this has been shown to generally be L2 [FSSN04]. Optimal buffer size can only be determined at run-time, because three factors vary between systems and during runs: memory architectures (number of cache layers, size, etc.), memory system contention, and stream rate and variability.

**Variable size** To automate run-time optimization of buffer-size, we introduce self-scaling ring buffers. These adapt their size at run-time based on “buffer pressure”: the distance between producer and consumer, normalized to buffer size. If pressure goes above a high-water mark a ring grows; if it drops below the opposite, it shrinks. We have implemented two types of scaling: ‘reallocation’ and ‘deck-cut’. Both are handled behind the interface, hidden from the user.

Reallocation replaces one I/O region with another of a different size. The left-hand side of Figure 4.7 shows how a producer can, instead of wrapping to the start of its current region, choose to allocate a new region and jump there. A reallocation operation can only be started when the producer reaches the end of the region. Buffer contents are not copied; As long as consumers are accessing the old region, both regions must be kept in memory. The approach is similar to rehashing and has the same drawback: during reallocation the buffer takes up more space than before.
Deck-cut avoids this problem. It allocates a maximum-sized buffer, but can temporarily disable parts of it, similar to how a deck of cards is cut: everything behind the cut is left unused. The right-hand side of Figure 4.7 shows a buffer with cut: the grayed containers are temporarily left unused. When cut, the memory region is unchanged, but the ring is artificially constricted. To maintain index validity, the cut can only be moved when the producer reaches the end of the ring. We will explain the index computation algorithm shortly. Contrary to real card-cutting, the algorithm does not involve choosing the spot arbitrarily; the same pressure variable as above directs this choice. Resizing is cheaper for deck-cut than for reallocation, because the only required action is to move the pointer indicating the start of the ring. As a result, it is well-suited to highly variable conditions. We exploit this characteristic by moving the watermarks closer together. A drawback is that deck-cut never returns memory to the general allocator. It places a higher continuous strain on memory area, but this impact can be reduced by granting temporarily unused memory to the disk cache (from which it can be reclaimed instantly when needed).

Read and write pointers must be monotonically increasing numbers (i.e., they may not be reset during a wrap), because the algorithm needs to be able to discern in which loop through the buffer – and in the case of reallocation in which I/O region – an index falls. To learn the offset of a block in an I/O region, the modulo of the number of slots in the ring $S$ is computed. When a buffer scales, $S$ changes. To guarantee correct offset calculation for all sizes, modulo operations must always overlap. In other words, all values of $S$ must be natural multiples of the same base. The higher the base, the faster the buffer expands and contracts (we only use base 2).
Figure 4.8: Effects of scaling at run-time

Figure 4.8 compares copy (i.e., write followed by read) throughput for a static buffer of 16MB with that of rings that gradually self-scale down from 16MB until they stabilize. The left-hand figure plots buffer size against time, where time is given in not in seconds, but in discrete loops through the buffer. It shows that the buffers gradually scale down and that deck-cut scales further down than reallocation, as a consequence of the moved watermarks. The right-hand figure plots throughput against time. Instead of scaling linearly with buffer size, throughput shows three levels, corresponding to access from main memory, L2 and L1 caches, respectively. The increase in throughput between main memory and L2 is significant: a three-fold improvement.

4.5.2 Allocation Method

Dynamically allocated buffers, where each container is individually allocated on demand, are ill-suited to the shared memory model of the BMS, because they cannot be mapped into memory protection domains in a single action. They hold one advantage over contiguous buffers, however: they conserve space. We have argued before that the memory wall renders space optimization an increasingly losing strategy, but in corner cases it is still the right one even on the fast path. On small devices and in times of extreme memory pressure, execution at reduced rate is preferable over failure. Dynamic buffers enable execution under such severe space bounds, if and only if all buffer clients share the same memory protection domain.

Previously [dBB08a], we stated that dynamically allocated rings are considerably less efficient than static regions. Figure 4.9 compares three static rings to two buffers based on malloc. It plots throughput against call size and shows that, indeed, throughput is considerably higher through all static rings.
Additional cost comes from the allocation and deallocation operations as well as from pointer walking during read and write to reach the right container. On second examination, however, we found that the read and write cost can be greatly reduced by caching the last pointer in the buffer interface, reducing the number of required lookups from \( \frac{1}{2}N \) on average to 1. The remaining allocation cost is not zero, but no longer significant. In other words, the figure shows results for our – inefficient – implementation, not for dynamically allocated rings in general. In shared buffers, the common case, static regions will still be faster, because access in these buffers by the various clients will not be purely sequential. In the dynamic case, this behavior will therefore still take a number of memory lookups per call.

### 4.5.3 Layout

Streams can be divided into two classes: byte- and record-oriented. Analogously, we classify rings as one of \( b\text{-DBufs} \) or \( r\text{-DBufs} \). While byte-oriented streams are more generic, record-oriented streams have lower access cost and enable secondary optimizations. \( b\text{-DBufs} \) accommodate requests of unbounded length. If a request flows beyond the edge of the buffer, it wraps around and continues. Individual items cannot be discriminated once data has been written. \( r\text{-DBufs} \), on the other hand, introduce markers to discern discrete blocks, e.g., for Ethernet frames.

**Record Implementations** Different record implementations have different performance characteristics. Records can be implemented using slots or markers. Slots are simple but waste memory if not filled up. Markers ren-
under seeking expensive, but compress data. A hybrid approach combines both advantages. Buffers with fixed-size slots (fr-DBufs) have a high percentage of internal waste, because slots are tailored to upper bounds. In case of Ethernet frames slots must be at least 1514 bytes, while the majority of packets are much smaller. Minimum sized packets with the highest ratio of waste to data (up to 95% of space is unused) are quite common [CMT98]. Internal waste is avoided completely by switching to variable sized slots. In such vr-DBufs a length field precedes each block. On a downside, seeking is considerably more expensive. Also, if no valid index into the buffer is known (e.g., because a consumer is slow and the producer doesn't wait, a situation we will discuss shortly), all data must be discarded, because slots cannot be identified afterward. Both drawbacks, seeking cost and overflow handling, are overcome by placing the headers out-of-band, in a separate, smaller circular buffer. A small buffer fits more headers in a single cacheline, reducing seek cost. It integrates behind the interface the now familiar model of an IBuf plus DBuf. Apart from this, such a double ring buffer – or dr-DBuf – behaves just like a vr-DBuf. Figure 4.10 depicts the three optimized DBuf implementations alongside a traditional continuous buffer that lacks markers.

**Results** We now return to Figure 4.9, which compares throughput of the three implementations with that of two dynamic allocation strategies. The figure shows that fr-DBufs indeed suffer from internal waste. For maximum-sized IP packets fr-DBuf performance is in line with the others. But as packet size shrinks, so does its relative performance: it is up to 30% slower than the other two. vr-DBuf and dr-DBuf results are on par. This was to be expected, as the advantages of dr-DBufs (seeking) are not evaluated in this test.

Figure 4.11 shows the impact of buffer size and per-call block size on throughput for slfile_read and slfile.peek calls. We show only an fr-
is inevitable (although which data has to be dropped is an open issue).

...data blocks at either the head or tail of the ring. When handling data ar-
than for the single user case. With a single producer and consumer pair,
Overflow detection and resolution with multiple consumers is more com-
plex.

4.5.4 Synchronization

Unix pipes define a strict and simple synchronization protocol. All producers
and consumers are serialized to use a single read and write pointer pair and
both block. In Streamline, buffers allow other forms of group synchroniza-
tion, complicating synchronization. It has a series of synchronization meth-
ods that selectively enable concurrent reading and writing at different data
loss and performance trade-offs. All these specializations in essence project
a set of (read or write) pointers onto a single scalar value. They differ in the
scalar they affect and their method of projection. A separate choice consists
of whether or not to block when synchronization is lost. The BMS supports
both methods. By default it follows UNIX pipe semantics. Somewhat simpli-
fied, this means blocking on a full or empty ring and unblocking early when
the other end closes the ring.

Multiconsumer Synchronization  Statically allocated rings can overflow.
Overflow detection and resolution with multiple consumers is more complex
than for the single user case. With a single producer and consumer pair, the
stream rate can be throttled in two ways: by blocking the producer or by drop-
ping data blocks at either the head or tail of the ring. When handling data ar-
riving from external sources, input quite cannot be throttled, thus dropping
is inevitable (although which data has to be dropped is an open issue).
The multiconsumer specializations of Streamline coalesce multiple read pointers onto a shared pointer to which the producer synchronizes. The implementations vary in how they coalesce pointers, with different effects on performance and reliability. We present three strategies: Slow, Medium and Fast Reader Preference (SRP, MRP and FRP). These methods drop blocks, thus they may only be used where this is acceptable – most notably in the network stack. As its name implies, MRP logically sits between the others. We will first introduce the extremes and then show MRP to be nearly as fast as (and to scale with) FRP, but to give the data correctness assurances of SRP.

**Slow Reader Preference: Safe** SRP is commonly known as non-blocking tail-drop. It silently drops all write requests during overflow. It is called *slow reader* because it must recalculate the position of the slowest consumer before every `slfile_write` call to know whether data must be dropped. Because calculation involves a sort over all read pointers, SRP scales worse than linearly \((n \log n)\) with the number of readers. For many applications, \(n\) is quite small. Still, even with low concurrency, the slowest reader determines the throughput of all others, which is unacceptable in a multiprogramming environment.

**Fast Reader Preference: Fair** To circumvent both issues we developed FRP, which blocks nor drops: a write always succeeds. Instead, consumers must compare their read pointer to the write pointer after processing a block to calculate whether the item has been overwritten in the meantime. We place the burden of guaranteeing correct behavior on the consumer, but assist with the `slfile_check` function that cheaply verifies pointer integrity after use. The safest operation is to use `memcpy` and then verify success. For this check to succeed, indices into the ring must be absolute and not be truncated to ring-length. Resolution then becomes trivial: if the consumer is behind the producer more than a complete loop, its position is moved forward. FRP is more fair than SRP, because only tardy consumers are punished. Also, with FRP, producers do not have to keep consumer state. Therefore an FRP buffer is easily shared read-only and without feedback channel. This is especially useful across high-latency links. On the other hand, FRP may aggravate operation under overload conditions, because it allows producers to continue to write – and thus increase load – even though this data can never be processed by the consumer(s). Also, because an FRP-enabled buffer is completely lock-free, data can be overwritten while a consumer is accessing it. For real-time applications or programs that can detect and repair transmission errors at a higher layer (e.g., Bittorrent clients), occasional data corruption at this layer
Medium Reader Preference: Balanced

Medium Reader Preference (MRP) can combine the positive features of FRP and SRP, if one extra assumption holds. The model is simple and centered around a single shared read pointer \( R \) that is updated by each consumer, who also maintains a private read pointer. Producers treat the buffer as SRP with a single read pointer \( R \). Whenever a consumer is scheduled it consumes \( n \) bytes and it updates \( R \) to its own previous read pointer if that is larger than \( R \) (this requires only an atomic test and swap operation). Assuming consumers are scheduled round robin, all other consumers now have one chance to read these \( n \) bytes before the first consumer is scheduled again. Functionally, MRP mimics SRP except for very tardy readers. Performance-wise it resembles FRP, because slow readers cannot hold up fast readers and \( R \) update calculation is cheap. MRP can only be used if consumers are scheduled more or less round-robin, whereby flexibility depends on the size of the ring. In other cases major data loss may occur. In many practical situations, fairness among clients is ensured by the scheduler and MRP can be safely employed.

Pointer Coalescing

Memory protection complicates the implementation of a shared pointer agreement protocol. FRP buffers run completely asynchronous, but SRP and MRP buffers must coalesce pointers. In MRP, projection is strictly a consumer task, since each read operation may result in an update to \( R \), but no write ever changes it. Consumers may not modify the shared value \( R \) directly, if they cannot mutually trust each other. Because updates to \( R \) will usually not occur with each read, they can be implemented reasonably efficiently with a kernel mode switch instead of shared memory (although in the reference implementation an unsafe direct access is used, instead). SRP synchronization is yet more difficult, because it executes a sort over all consumer read pointers, which demands read access to all their private metadata segments. Whether the sort is executed by consumers or by producers...
Ring Types

should be chosen based on relative access frequency. To minimize cost, read-
m ost buffers should sort during writes and write-most buffers the other way around. This is one specialization vector that we did not pursue further. For both MRP and SRP, the safe and fast update solution is for each consumer to have its metadata segment shared read-only with all clients. This solution costs one memory page per client only for communicating a counter. This is acceptable for rings of at least a hundred pages. The kernel mode switch method is used for smaller buffers (the exception).

Locking  Locking cost is low to non-existent for these multiconsumer extensions. FRP operates completely lock free, MRP only requires atomic operations on scalars. The sort operation performed by SRP, on the other hand, is an example of processing that naturally breaks the lock-free nature of rings, because correct output of the sort algorithm can only be guaranteed with mutual exclusion. Regardless, and perhaps against intuition, lock-free operation is safe from deadlock and data corruption. To understand why, we must investigate the exact effect of race conditions on operation. The sort operation selects the threshold for the producer to synchronize on. A race between the sort and a private read pointer update will result in selection of a stale pointer, causing the producer to wait unnecessarily. Deadlock will occur if the operation never unblocks and consumers are blocked at the same time. When sorting is performed by consumers, deadlock cannot occur, however. Each consumer first updates its private pointer and then executes the sort. With a single consumer, a sort trivially produces a correct result. With multiple consumers, some may read a stale value. If they manage to update the shared pointer last, this causes a producer block at an incorrect index. In the unlikely event that this stale index is an entire ring off, the block coincides with a consumer block to cause deadlock. One consumer will always see the latest (correct) state, however. If the shared pointer may only be increased and the required combination of comparison and update is implemented with an atomic test-and-set instruction, this correct consumer will always produce the definitive shared read pointer. Thanks to the atomic test, even if another consumer was preempted during its sort and returns later with an incorrect overly conservative result, the shared read pointer will be correct and deadlock is avoided. This method relies on the algorithm storing the lowest ring index observed during a sort, not a reference to the currently slowest reader, because the reader may update its value during the sort, causing another to become the slowest.

In practice, deadlock is further avoided as a result of event batching. Even if a producer would sleep on an incorrect value indefinitely in principle, this
Communication

is avoided in practice because the timer signal (explained in Section 5.2.2) periodically wakes up each waiting process. Moderation increases efficiency at high rates, but causes background noise at low rates (by choice, to curb latency), precisely when deadlock can occur. Occasionally waking up all blocked clients will initiate forward progress by at least one during deadlock, as the 'queue not full' and 'queue not empty' expressions will both be checked and cannot both be false at the same time for a queue of non-zero length.

Application The rules for which multiconsumer mechanism to use are simple. Tail-drop must be used wherever drops are forbidden, e.g., for pipes. For network traffic, rules are more relaxed as overflow along the I/O path is indistinguishable from network failure. Here, we generally use FRP. The only exception is when a ring is shared read-only between userspace processes, such as a network reception queue. MRP then performs better because it is protects processes from one another like FRP, but also throttles traffic at the input like SRP, and so gives feedback to congestion control protocols.

Multiproducer Synchronization Some tasks, such as a network transmission queue, can have multiple processes producing data concurrently. Mutual exclusion can serialize these requests, but is inefficient if parallelism is high. Like multiconsumer access, multiproducer access benefits from a more efficient implementation. There is prior work on concurrent updating of queues (e.g., [MS96]), but these approaches protect only the atomic enqueue operation. They are practical when handling dynamic pointer structures, but not in data rings, where an enqueue operation involves non-atomic copying. Here, concurrency is served best when that operation can be parallelized. The extended Unix interface provides the optimistic poke call and its asynchronous pair for this purpose.

Locking Writer-writer conflicts can occur when multiple acquisition calls are made while the write pointer is updated in the completion call. Concurrent production can be implemented with minimal locking at the cost of storing two bits per concurrent user in the shared metadata element. The first bit encodes whether a container has been opened for writing, the second whether it has been closed again. Together, $N$ such pairs of bits form a bitmap. The first pair encodes the state of the container just beyond the write pointer; each successive pair encodes state for the next container. As it operates on containers, the approach cannot be feasibly employed on byte-oriented streams. Acquisition calls may return successive records up to the
read pointer or until \( N \) is reached. Initially, all bits are unset. Each successive acquisition call sets the next unset even bit (0, 2, 4, etc.). Each completion call compares its position to the write pointer to calculate the bit pair corresponding to its container and sets the odd bit, so that both are now set. If all bits are set up to and including its pair, the call grabs a writer lock, calculates the number of consecutive set bits, shifts all out of the bitmap, increases the write pointer by half that number and releases the lock.

4.5.5 Replication

To efficiently integrate peripheral resources, Streamline must apply copy, context switch and cache invalidation reduction techniques not just between processes and the kernel, but across peripheral buses, CPU interconnects and possibly even network links. Latency and bandwidth characteristics of these media differ substantially. Across peripheral buses and network links caching, prefetching, burst and hardware (DMA) offload can and must be employed to minimize the gap with local memory. The BMS optimizes access behind the Unix interface without client interference, by replicating data on the consumer side of a slow transport medium. We call interfaces that optimize high-latency access *shadow* buffers. The diversity in media bottlenecks has led to four shadow buffer types: zero copy, copy once, hybrid and prefetching.

*Zero-copy* accesses the imported data segment as if it were local. It uses the same read and write implementations as non-shadow buffers. Zero-copy is efficient for shared memory communication, such as between kernel and userspace. When bandwidth is narrow, latency is high, or features such as burst mode or DMA offload can be exploited, zero-copy is suboptimal. Across the shared PCI bus, for example, bulk one-way transfers benefit from both burst mode and DMA offload. While zero-copy access is possible, every read incurs a round-trip delay. For high-speed sequential access—the common case in I/O processing—copying a block using hardware support (*copy once*) is more efficient. This method does not directly reflect read call requests to the imported buffer, but keeps a local shadow copy of its imported data segment and issues bulk copy requests between the two. In networking, especially for forwarding and filtering applications, it is common to initially only inspect packet headers and based on this to decide whether to access the larger payload. Both zero copy and copy once are inefficient for these applications, one in the common case of header-only processing, the other in the more expensive case of payload inspection. A *hybrid* solution supplies the first \( n \) bytes of a block directly from an imported segment in zero-copy fashion and switches to copy once bulk transfer for payload requests. For TCP/IP
Table 4.3: distributed memory access methods

<table>
<thead>
<tr>
<th>Access Method</th>
<th>Advantage</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero copy</td>
<td>Copy reduction</td>
<td>Low latency links</td>
</tr>
<tr>
<td>Copy once</td>
<td>Efficient copying</td>
<td>High latency links</td>
</tr>
<tr>
<td>Caching</td>
<td>Bimodal access</td>
<td>Header processing</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Latency hiding</td>
<td>Very high latency networks</td>
</tr>
</tbody>
</table>

networks the threshold is hardcoded to the default payload offset. Finally, on high-throughput, high-latency links such as WANs, sequential throughput benefits from speculative copying of more data than is directly requested, or *prefetching*. Table 4.3 summarizes the four methods.

**Application** All non-programmable peripheral devices are accessed through the copy once interface. Programmable devices are instructed to push data directly into host buffers using DMA, instead (if they are capable of bus-mastering). Across network links and for disk I/O prefetching is the default, between kernel and processes it is zero-copy.

Related to distributed transfer is distributed synchronization. Across the PCI bus, or more generally speaking across buses with burst capabilities, frequent master switching kills performance. Here FRP synchronization performs much better than its alternatives. As no state has to be fed back from consumers, the producer can keep a continuous (for all practical purposes) lock on the bus. FRP is always used to forward unacknowledged data across half-duplex or otherwise contended media.

### 4.5.6 Device Access

Many streams originate or terminate at peripheral devices. Without optimization, a single transfer between devices incurs at least one horizontal copy between the peripheral subsystems (e.g., disk and network) and more commonly two vertical copies between kernel and userspace, with associated context switches. To avoid this unnecessary cost, Streamline incorporates device interfaces that directly communicate through Beltway Buffers. It exposes two interfaces: a storage interface for block devices and a streaming interface for network and peripheral devices.

**Block devices** IO-Lite [PDZ00] showed that an integrated BMS can remove copy overhead to and from the disk cache. Streamline extends IO-Lite
by making these savings available to legacy applications through transparent splicing. Unlike IO-Lite, Streamline cannot as yet splice to the cache, because it relies on the Linux native disk cache which demands full control over its contents. Streamline enables read-only splicing from the disk subsystem by exposing cached files as DBufs. Like DBufs, file contents are locally accessible both from kernel- and userspace. In the kernel, Streamline reflects calls to the Linux virtual filesystem layer. Exporting the page cache to userspace is more involved. The advantages for splicing here are clear: it reduces context switching and allows high-performance transfer without moving logic to the kernel. During the open call, Streamline sets up virtual memory mappings for the complete file and the OS page-fault handling mechanism maps in the uncached pages on demand. Because all mappings are created during open, access control checks are restricted to this single call, improving runtime performance. In essence, the presented block device interfaces makes a well known performance technique – memory-mapped file access – available to all legacy applications.

Application  Generic file splicing extends copy-free transfer to non-trivial servers. Our motivating example is a webserver that serves dynamic content by executing server-side scripts. Scripts often only combine blocks from multiple static files. The canonical example is combining a site-wide header and footer with a URL specific body. Currently, servers must cache these blocks manually to reduce disk I/O overhead. Transparent splicing offloads caching concerns: servers can satisfy page requests simply by writing a sequence of indices to the network transmission queue, reducing application complexity and memory footprint. Userspace page cache access enables splicing from the disk cache. This generalizes sendfile, a system call found in many OSes that speeds up static file serving by handling all data movement within the kernel. Because file serving consists of looping over a read from disk and a write to a socket, file server processes see increased throughput from offloading this loop to the kernel: it removes many usermode switches. The sendfile interface can only transfer single file regions at once and still needs to cross the ABI for each region, however. Exposing both the page-cache and network transmit queue to userspace allows Streamline to splice arbitrary data blocks with an IBuf write as only cost, removing all context switches.

Network devices  The network interface, like the disk, is a critical I/O-path endpoint whose throughput must not be constrained. In general, Beltway communicates with network devices through two rings: a reception DBuf and a transmission IBuf. The heart of a Beltway device driver is made up
of two specialized buffer implementations whose data layout matches the hardware specification of the device reception and transmission rings. For a simple interface, the reception DBuf is passed on to a software processing pipeline that exports intermediate streams through IBufs on request. Smart NICs that perform protocol processing (e.g., checksum calculation and protocol classification) can register these device features as Streamline filters to replace parts of the software pipeline. On the transmission path, additional functionality is also registered as filters.

**Zero-copy Networking**  
Network streams are traditionally copied between the kernel and user processes. The BMS replaces, subject to access control, this slow operation with *zero-copy network reception*: it maps device buffers directly into a process's virtual address space. Section 6.5.2 discusses the safety aspects of this approach. On the transmission side, we must be hesitant to apply direct access because it grants users the right to change contents after a send operation returns. This, in turn, opens the path for malicious users to send illegal packets out on the network or to craft packets that destabilize the kernel or network interface. Data corruption is not critical in itself when rings are unique to a process or user, because a process can only harm itself in those cases. System destabilization should not be possible to begin with and is correctly fought through defensive coding, not memory protection. Blatantly obvious malicious uses, such as spambots and DoS attacks can be spotted with kernel-level monitoring filters (stream sanitization, discussed in Section 6.5.2) and contained with strong resource control. Because more stealthy attacks are not so easily discovered, transmission queue mapping must simply *never be allowed* for unprivileged users on (semi-)public machines, such as workstations in cybercafés, libraries and University laboratories.

### 4.6 Summary

This chapter introduced the buffer management system that governs data transport in Streamline. The BMS minimizes the number of copies, data-cache misses and task switches for a generic stream processing system that spans across devices, kernel tasks and user processes. It reduces cost over a straightforward store and forward queue network in three ways. First, it replaces private data copies with shared memory. To amortize memory allocation and virtual memory mapping cost, it scales containers to large multi-block rings (4.2). Second, it adds an optional layer of indirection to avoid copying. For legacy applications it enables transparent indirection, known as
splicing (4.3). Finally, the BMS specializes buffer implementations to tailor buffers to application and system. Because some specializations are orthogonal, the BMS supports implementation stacking (4.4). This chapter introduced specializations for saving memory (and cache misses) through resizing, dynamic allocation and data layout as well as others for group synchronization, replication and device integration (4.5). Together, the ideas presented in this chapter contribute to the central goal of a high-throughput, flexible I/O system by substantially reducing the base cost of communicating large datastreams between many filters in many environments. A generic streaming I/O communication layer renders the use of pipelined I/O feasible for networking applications.