Beltway buffers: avoiding the OS traffic jam

Willem de Bruijn
Vrije Universiteit Amsterdam
wdb@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@few.vu.nl

ABSTRACT

Beltway buffers are operating system I/O paths optimised for high-throughput network applications. The key architectural feature of *Beltway buffers* is that all I/O takes place in long-lived, allocation-free, shared ringbuffers. Advantages of this design are (1) improved throughput through system-wide copy, context-switch and allocation avoidance and judicious use of the data cache, (2) transparent integration of peripheral hardware and (3) simplicity and familiarity due to comprehensive use of the POSIX file interface for accessing streams.

I. Introduction

We introduce a novel system-wide buffer management system based on extensive use of ring buffers. The system provides all well-known primitives (sockets, file descriptors and pipes) to support legacy applications. In addition, it carries advanced I/O features (e.g., user-visible disk caches, multi-way named pipes, copy avoidance and splicing, prefetching and interrupt mitigation) that improve application performance. Internally, *Beltway buffers* are structured quite differently from existing systems.

Traditional network stacks incur cross-layer copying and task-switching overhead as a result of strict compartmentalisation. By default, reading data from disk and writing it unmodified to the network involves two privilege escalations to handle disk I/O and networking, as well as two copies across the kernel/userspace barrier. For this most basic example, the `sendfile` system call brings relief, but for the majority of more complex I/O interactions it is too simplistic. They must resort to application specific band-aids. Dynamic webservers, for example, duplicate filecaches in userspace.

Only a small subset of the systems in use need the strong security guarantees enforced by strict task separation. Instead of incurring this cost on all applications and trying to add exception clauses for certain uses, we propose to remove the restriction in general and only impose layering for those applications where it makes sense. In practise, this means improving performance for the vast majority of machines that operate as network servers or single-user workstations. Separately, we shall show that this architecture can offer the strong protection demanded by multi-user systems.

Beltway buffers explores an extreme in the design space for communication paths: where most existing systems are careful not to waste memory, we sacrifice RAM to gain performance. This design point is increasingly viable as processing and memory density costs decrease faster than access latency and, consequently, datarates.

This paper makes the following contributions:

1) It presents a novel operating system I/O architecture with attractive properties for network applications.
2) It identifies causes of avoidable I/O overhead in traditional network stacks by comparing the novel architecture directly against a widely used alternative (Linux).
3) It explains how the novel design can cleanly integrate existing and upcoming networking hardware (such as programmable and multi-ring NICs).
4) It discusses implementational issues involved in building a practical high-performance I/O stack and presents solutions.

These abstract contributions have direct practical benefits, such as support for copy-free data movement (known as 'splicing'), multi-point pipes, zero-overhead network monitoring, and vastly improved transfer rates even for unaltered POSIX file I/O operations.

In Section II we show how *Beltway* is used in a common OS with real applications. Section III contains related work. Section IV discusses the *Beltway* design, while Section V shows how buffer implementations vary for optimal performance. Section VI discusses how the buffers are used to provide abstractions like sockets, pcap, etc. In Section VII we evaluate our work and in Section VIII we draw conclusions.

II. Applications

To demonstrate how the existing situation limits networking performance, we introduce two very different networking applications and their current implementational weaknesses: A file/web server (FS) pushing data from disk to the network, and a network monitor (MON) that sniffs incoming traffic. **FS** and **MON** will be used as running examples throughout this paper to explain the practical impact of abstract innovations.

Fileservers are interesting because they have two potential bottlenecks: disk access on the back-end and network access on the front-end. Additionally, the hand-off between the disk cache and the network stack often introduces superfluous copying. Indeed, this issue is so common that the `sendfile` system call can be found in many operating systems. But `sendfile` is useful only for a narrow subset of server tasks: serving of complete, unaltered, files. Recent versions of Linux support splicing to transfer data between kernel subsystems without resorting to copying [1]. This implementation adds a system call and demands application support. As we will show,

the *Beltway* design trivially allows splicing and also reduces the other two bottlenecks.

Network monitoring applications show a different source of waste: unnecessary copying and task switching during packet capture. Monitoring forms the basis for critical security tasks, such as intrusion detection, but the present high capture cost prohibits live monitoring of production machines.

While this is the first time that we submit a description of the *Beltway buffers* for publication, they were used 'under the hood' in several projects. For instance, We used them to implement high-speed intrusion prevention on embedded hardware [2].

III. Related Work

Copy overhead in network stacks is a well known problem. Copy-avoidance mechanisms have been proposed that replace copying with page remapping to reduce overhead. Brustoloni [3] categorised previous efforts and showed them to perform roughly identical. Druschel *et al.* describe copy avoidance ideas for network buffers [4] and subsequently translate these into Fbufs [5]: copy-free communications paths across protection domains that can sometimes amortise the per-block costs. Fbufs are only efficient if mappings can be reused, for which early demultiplexing is required. This is also true for container shipping [6]. Fbufs are later incorporated in IO-Lite [7], which introduces mutable pointers to immutable buffers to replace copying system wide.

In contrast to these projects, *Beltway buffers* eschew per-block allocation; they amortise allocation overhead by placing all blocks into large, reusable ring buffers. Govindan [8] introduced memory-mapped ring buffers to reduce cross-space communication. He used buffers as one-to-one pipes between applications and kernel. We extend this basic notion by allowing *shared buffers* between *arbitrary* address spaces, be they applications, OS subsystems, or embedded logic. Furthermore, we implement sharing throughout the system: arbitrary sets of buffers can be shared between arbitrary sets of processes. The resultant buffer subsystem is generic: tailored to networking, but also applicable to other cross-space communication such as IPC. Ring buffers can already be found in most network stacks, for instance U-Net [9] and the Arsenic network interface [10], but we believe that we are the first to extend the idea beyond the scope of a single path to a generic OS communication layer.

IV. Architecture

Accepted operating system design principles add unnecessary cost to I/O, especially in the case of network traffic. Conservative memory protection limits the ability to share data. Additionally, monolithic operating system kernels add cost through compartmentalisation: separate subsystems for networking, disk I/O and system call handling duplicate code (e.g., for caching), introduce buffering (which increases latency and decreases cache efficiency) and implement local optimisations at the detriment of global throughput.

A well-structured system-wide I/O architecture increases data throughput in two ways: (1) it removes inefficiencies emanating from the hand-off between subsystems, and (2) it makes complicated performance optimisations cost-effective by applying them globally. *Beltway buffers* aim to provide just that in a popular OS (Linux), where they underlie all I/O mechanisms (sockets, pipes, disk, etc.). The design revolves around moving all data transport throughout the processing stack (process, kernel, peripheral devices) into shared ring buffers.

A. Advantages of rings

Most operating systems implement per-block dynamic allocation: a memory region must be acquired for each data block. Pointer queues are maintained to structure streams (e.g., into FIFOs). In contrast, we statically allocate (a) *shared* data rings (DBufs) capable of holding multiple data blocks, and (b) *private* index buffers (IBufs) with pointers into these rings (Figure 1). Rings hold a number of advantages over dynamic allocation with pointer queues:

- *Amortised Allocation.* Static allocation amortises cost across many blocks.
- *Amortised Copy Avoidance.* Similarly, remapping operations are carried out once per process lifetime, not for every block.
- *Rich pointers.* Pointer queues are not valid across address spaces. Furthermore, pointers provide no context (e.g., TCP flow, or movie frame), which causes them to be embedded in complex shared datastructures that demand locking. In *Beltway*, each consumer has an *IBuf*, a private set of *rich* data pointers, valid across address spaces. IBufs are lock-free and yield good cache behaviour through their reduced size.

If rings have such obvious advantages, why have they not been used more extensively before? There are a few obvious challenges to implementing rings effectively. For one, they trade memory utilisation for speed. In the past this was unacceptable, but as RAM size growth outpaces access time the option becomes increasingly viable. Furthermore, memory waste can be curtailed. Another issue is that rings are coarse-grain structures: they do not enforce per-block security policies. The remainder of this paper – and indeed the core of the *Beltway buffers* – deals with overcoming these obstacles.

B. Collections of Rings

Battling complexity, a single ring would be ideal. Unfortunately, that fails even for the simplest of tasks. Take a MON-like application in which drivers receive all data in a single, shared ring buffer that is shared between all applications. Such a ring incurs no copies and has no runtime allocation. Unfortunately, the solution fails for multiple reasons.

First, unacceptable security concerns arise when there is no control over who has read and write access to data within the ring. Unprivileged users' MON applications, for example, may not inspect any traffic but their own. The authors

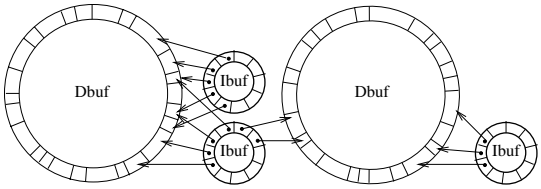

Fig. 1. Chain of data rings with data and index buffers with pointers to data

of FBufs [5] have previously voiced these concerns about memory architectures with coarse-grain security enforcement.

Second, in the presence of multiple logical processors, a single ring is shared by all processors which leads to cache conflicts. In contrast, giving each processor its individual buffer limits working-set overlap and thereby increases throughput. We want to avoid silly cache behaviour, whereby an update by one consumer leads to a cache invalidation for another even though no data dependency exists. The same holds for metadata updates by producers and consumers. Similar objectives led to Van Jacobson's 'netchannel' [11] architecture where the processing of a packet, including TCP, is tied to a single processor. We facilitate the same behaviour.

Third, besides multiple cores performing end processing, we must also attend to embedded resources. For instance, NICs may run some functionality on the card (filters for MON, say, or checksumming for FS) using either on-board buffers [2], or buffers in host memory [12]. The I/O architecture must support non uniform memory access, reconcile distinct memory addressing schemes, integrate varying hardware ring implementations and select globally optimal buffer placement policies.

Finally, transformation of data (such as TCP reassembly, encryption, etc.) may be hard to do 'in-place' because of security or buffer space limitations. In that case, data may be better duplicated in a different buffer.

C. Data Buffers and Index Buffers

At the heart of the *Beltway buffers* are DBufs and IBufs of varying sizes and use-cases (Figure 1). DBufs can contain arbitrary and possibly mixed data, of both discrete (e.g., IP packets) and continuous (e.g., a UNIX pipe) nature. Each IBuf belongs to a consumer: a kernel task or application interested in a subset of the data contained in the DBuf. An IBuf represents a consumer's view on this data. For example, a media application's IBuf may point to the start of all frames in a videostream.

An IBuf may contain pointers to multiple DBufs and multiple IBufs may point to the same or different blocks in the same DBuf. Figure 1 depicts the varied relationships. Both kinds of freedom are necessary. The first is used when a consumer accesses data arriving over multiple paths, e.g., a MON application monitoring two NICs. The second case occurs with multiple processors. Here, data consumers may be scheduled on separate cores. By maintaining the data list for each consumer in a separate IBuf, interference is minimised. Producer and consumer metadata is mapped in different cachelines, so updating the write pointer does not invalidate the consumers' cachelines. In addition, in many demanding applications, like network processing or graphics,

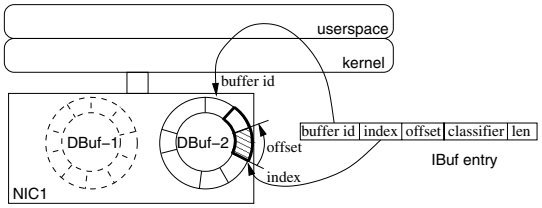

Fig. 2. IBuf entry points to a byte in a specific DBuf on NIC_1

consumers process disjoint sets of data blocks. For instance, applications process their own set of IP packets or frames. By aligning buffer slots on cacheline boundaries, we decouple data access on multiple cores.

D. Allocation and Memory Mapping

Beltway buffers minimise copying through aggressive sharing. In most cases, a block is copied into a DBuf once and accessed by everyone from there. For instance, FS applications read from a file-cache and only push indices to the transmission queue, where they are combined with headers, scatter-gather style. MON applications directly snoop other processes' DBufs to filter in-place. *Beltway* minimises copying both *vertically* (e.g., between NIC, kernel and applications for, say, FS sending data to the network), and *horizontally* (e.g., demultiplexing the same data to multiple applications for, say, a MON application sniffing incoming traffic). As calls to the general memory allocator are expensive, buffers are allocated in advance as a contiguous virtual memory block, e.g., during application initialisation. Because the entire buffer is a single block of memory, it can be mapped into a protection domain in a single operation.

Buffers encapsulate data into what we term *soft segments*, the software equivalent of hardware-supported memory segments. Ideally, they should be implemented using hardware segments, but the current implementation uses a set of *pages* protected by the MMU.

Setting up and tearing down shared memory (e.g., using the POSIX `mmap` call) is expensive. Per-page copy avoidance mechanisms incur this cost for each block. Even though the benefits outweigh these costs, profits are not maximised. In our shared rings, mapping costs are amortised over all blocks and consumers, rendering them irrelevant.

E. Indirection Details

As DBufs may reside in different protection domains and the *Beltway* system does not assume a single address space, IBuf elements have to be valid across all address spaces. For this reason an IBuf entry contains a 3-tuple that serves as a pointer: a globally unique identifier of the corresponding DBuf, an index into that buffer to select a block of data, and an offset into the block. Prior to accessing, say, a network packet for the first time, the IBuf pointer is translated into a local pointer that is cached for subsequent accesses. An example IBuf entry is illustrated in Figure 2. It provides a global pointer to a byte in a buffer on one of the network cards.

IBufs also contain a length field to complement the pointer, as well as a field to store classification results. For instance, an

IBuf producer in MON can be a classifier that weeds out non-TCP traffic and stores the destination port in the classification field. If the next processing step (e.g., a filter or transcoder) selects packets based on classification results only, no accesses to DBufs are required at all. Only IBufs are accessed, which is attractive for cache behaviour. In practise, we found that the offset and classifier fields in IBufs are useful when handling encapsulated data blocks corresponding to byte streams (such as TCP segments or movie frames). For instance, the index may point to the start of a TCP segment, the offset to the data, while the classifier stores the flow id. In Section VI-0c we discuss in-place TCP reassembly for MON along these lines.

When reading from or writing to a DBuf, calls directly access the local data ring. When accessing an IBuf, however, we do not return the IBuf contents. Instead, we silently resolve the corresponding DBuf and return a block from there (provided the IBuf pointer is valid). Transparent indirection makes application programming simpler, because it gives applications a private view on data without needing a separate interface. If the DBuf referenced from an IBuf element is not accessible in the current address space it is silently mapped in, in a manner reminiscent of page-fault handling.

Beltway buffers can be viewed as a second virtual memory layer with an unconventional addressing scheme. One might object that it is simpler to build queues with pointers. An IBuf index lookup is fairly expensive because a locally valid address must be recalculated. The benefits of our approach are that indices remain valid across address spaces. We do not presuppose a single address space system, as that would limit applicability. It is also not necessary, because overhead can be amortised to reduce cost.

F. A Uniform, POSIX-based Buffer Interface

No ring buffer design is optimal for all combinations of tasks, domains and users. Some tasks process data in fixed-size segments, while others access continuous byte-streams. Between some protection domains memory regions can be shared, while others leverage hardware support for pushing data across. Moreover, between domains differences in hardware alignment and endianness must be taken into account. Dealing with such buffer peculiarities usually leads to specialisation within the core code-path, the opposite of a structured OS. In contrast, *Beltway buffers* hide these implementation details behind a common interface. All buffers, anywhere in the system (even the kernel), look alike to the consumer. This choice has allowed us to freely experiment with novel buffer implementations.

Instead of defining our own interface, we build on the well-known POSIX file API. The POSIX API is used to directly access *all Beltway buffers*. By itself the POSIX file API is a convenient API for FS applications. On top of this we implemented other communication interfaces, most importantly BSD Sockets, UNIX pipes and the packet capture (PCAP) interface for MON applications (like `tcpdump`, `ntop` and `snort`). We will discuss these interfaces and their use of buffering in Section VI.

Our implementation of the POSIX file API is sufficiently flexible to allow multiple processes to share a single buffer to implement a multicast pipe, and also to implement efficient communication abstractions similar to the `splice` call proposed by McVoy [13]. `Splice` controls a kernel buffer from userspace. It allows data to be moved to/from the buffer from/to arbitrary file descriptors. FS greatly benefits from `splice` as disk data is no longer copied to userspace. MON programs benefit from it when saving interesting data to disk.

Following the POSIX convention, a *Beltway buffer* is opened using:

*open (const char *name, int flags [, mode_t mode [, size_t size]])*

The name argument is optional and can be used for named buffers (similar to named pipes). The fourth, also optional, parameter sets buffer size. `read`, `write`, `lseek` and `close` all follow convention. Semantics are also similar, but not identical. The two main differences between streams and files are that streams can be in theory of infinite length, and that only part of the stream (the sliding window) can be accessed at any time, whereas files are of fixed length and randomly-accessible [8]. This is reflected in the possibility of repeated calls for the same data failing in streams.

A known performance drawback of the POSIX API is that it is based on copy semantics, that is, `read` and `write` always create private copies of blocks for the caller. Such semantics are safe, but often wasteful, as they must be implemented using copying or VM modifications such as copy-on-write. For this reason, alternative communication methods have been designed, such as U-Net [9] and IO-Lite [7]. To circumvent these costs with only minimal changes to the application, we extend the API with only a single function: `peek`, an alternative to `read` that uses weak move semantics in the nomenclature of Brustoloni and Steenkiste [3]. With weak move semantics actors may read blocks, but not modify them, because blocks can be shared. Instead of making a private copy, the subsystem returns a pointer to the original location (somewhere in a DBuf). The function is identical to `read` apart from the second argument, which takes a double instead of a single pointer:

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

G. Security

These semantics are easily enforced between memory protection domains by sharing the pages underlying a buffer as read-only. This is one example of a more generic, unconventional, approach to access control afforded by long-lived shared buffers. In *Beltway*, each buffer has a single access control policy, Similar to POSIX file access, buffer access is restricted to an access control group (gid) or user (uid). Unlike files, buffers can also be shared only within process groups (pid). Protection is enforced per buffer and protection domain pair; access rights are checked only on the first access to a ring from a domain.

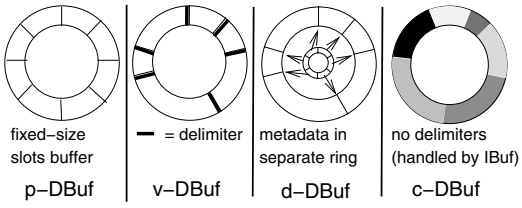

Fig. 3. Different implementations of DBufs

Coarsening of access control removes one type of runtime cost. *Beltway buffer* function calls are handled in the context of the caller, whereas traditionally calls must cross to the kernel to guarantee correctness. Making the API orthogonal to the user/kernel ABI saves context-switches and consequent cache invalidations. It trivially enables zero-copy network monitoring. More in general, it allows copy-free transport between groups of processes and devices. With multiple NICs (or multi-ring NICs) private, secure zero copy networking can be configured for multiple applications, users or groups.

The stricter POSIX semantics must still remain enforceable. In *Beltway buffers*, `read` is left unaltered. By default, `write` is backed by a statically mapped ringbuffer, which in principle allows applications to alter buffer contents after the call returns. When untrusted processes can harm others or the OS kernel by circumventing said policies, strict enforcement through memory protection (and thus context-switching) can be enabled on a case-by-case basis. With private transmission queues applications can only harm their own communication, against which no protection is necessary.

H. Disk access

Beltway buffers integrate the page cache (the cache for disk data managed by the OS) to enable reading from the cache without runtime privilege escalation.

In this scenario a single shared cache – as commonly employed – is not sufficient because of security reasons. Instead, each consumer may have three buffers for caching: a private cache, a cache shared by its group, and a cache shared by everyone. Just like other buffers in the *Beltway buffer* system, the cache is shared according to access rights attached to pid, uid, gid, or other. The disk cache can safely be made visible from userspace and shared with other consumers.

FS like applications often consist of a loop over a `read` from disk and a `write` to a socket. This scenario has been optimised through `sendfile` in many UNIX OSes. *Beltway buffers* have a more general solution that works between any collection of rings. By enabling what we call the **fast splice optimisation**, the two connected operations lead to `splice`-like functionality without requiring a change in interface. Using `peek` no data is even copied to userspace. With the optimisation, a write call compares its passed data pointer to the last block seen by `peek` or `read`. If they match, and the write is to an IBuf (e.g., for network transmission), instead of data the generally smaller IBuf element is copied. Even if a `read` is used instead of a `peek` administrators may configure buffers as having `splice`-like behaviour. Although the read incurs a copy, the write copy and user/kernel-mode switch

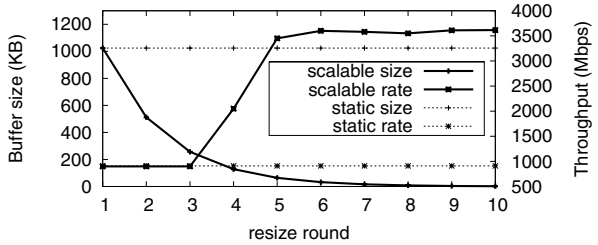

Fig. 4. Scalable buffer performance

are saved. As a result, even unmodified legacy applications experience significant speed-ups. This cannot be made the default behaviour, because it causes data corruption when a process modifies data. For FS applications like webservers it will be turned on, while for decoders it will be off.

V. Specialisation

We now show how modifying the buffer benefits throughput, drop rate and other characteristics. So far we have seen two types of buffers, DBuf and IBuf, which differ in task: the first hold variable sized blocks such as network packets, the second small, fixed-size indices. Could adapting the underlying buffer implementation to the use-case improve performance? Figure 3 shows a few example implementations. In this chapter we will show multiple, orthogonal, design trade-offs. Where applicable we note defaults for specific tasks. These can always be overridden through flags in the `open` call.

A. Increasing Memory Bus Throughput

To maximise memory throughput we leverage mechanisms such as prefetching, burst reading and, most importantly, caching. Optimising code for these features is complex, because cache hierarchies and memory technologies differ widely. We will not discuss ways to optimise code to a specific hardware environment. Instead, we consider a more general rule-of-thumb: *a decrease in runtime memory usage will lead to an increase in throughput*. 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 therefore not the goal. Furthermore, only waste that influences performance needs to be minimised. Most of the waste incurred by shared rings is due to statically allocated, unused memory, with no impact on cache utilisation. We discern two types of waste: external and internal.

1) On-demand Resizing: External waste is the unused part of the ring spanning from the head to the tail. Memory locality can be increased by reducing this gap, but a buffer tuned too well to the average case will overflow during bursts. The trade-off between memory locality and overflow is hard to make in general, but clearly some inefficiency is unavoidable when buffers are pre-allocated.

External waste can be limited by adapting the buffer size at runtime. To investigate this idea, we built a resizable buffer that —unlike heap-allocation— is contiguous and for which resizing is more costly than static operation. When resizing is requested, a new memory area is allocated (akin to rehashing). While there are references to the old area both must be probed.

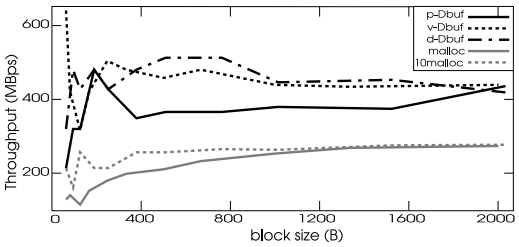

Fig. 5. performance of different ring implementations

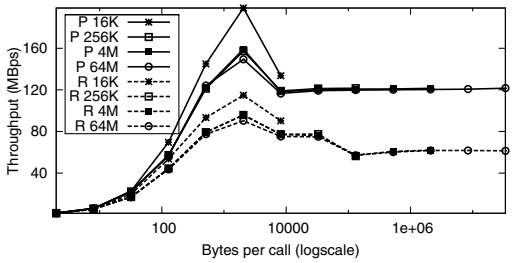

Fig. 6. Impact of ring size and datablock size

Thereafter the old area is released. Computational overhead is minimal, but memory pressure considerable during resizing. Buffer sizes are based on powers of two, therefore pressure is either 1.5 or 3 times as high as originally.

To circumvent the memory pressure issue, we have also implemented a *pseudo*-resizable buffer that does not change the underlying memory area, but instead reduces the *perceived* size. Only part of the ring is used, similar to how a deck of cards is "cut". The added logic makes calls more expensive than for a static ring. Both scalable buffers trade off computational complexity for an increase in locality of reference. We compared the pseudo-resizing buffer to a static ring in Figure 4, which shows throughput in consecutive rounds of resizing, as well as the size of the buffers. The sudden hike in throughput is a result of the sequential nature of ring access: for a static distance between consumer and producer, either the entire intermediate working-set fits it the cache, or LRU causes each consumer access to result in a miss. The resizable buffer was observed to perform similarly. Automatic runtime scaling is implemented using high and low watermarks on producer-consumer distance. Resizable buffers thus automatically adapt to cache and working-set size. Not visible in this figure is the computational overhead of adaptation. Unoptimised code did show up to a 10% overhead, but this disappeared with compiler optimisation.

2) Slot Compression: Internal waste is unused space within an allocated block. Traditional slotted buffers 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 [14].

Internal waste is removed completely by switching to variable sized slots. In such a **v-DBuf**, a marker denoting the length of a block precedes the block itself. Direct access is identical to access to a slotted buffer, but seeking is considerably more expensive, because a simple computation no longer suffices: markers must be read, which adds memory lookups. Also, there is no concept of a single loop in the ring. If no valid pointer is known, e.g., because of overflow, the only option is to update all the way to the shared offset, because there is no way to discern markers from data after-the-fact.

Both drawbacks, seeking cost and overflow handling, can be overcome by placing the headers out-of-band, in a separate – smaller – circular buffer. The small buffer fits more headers in a single cacheline, reducing seeking cost. Marker validation is identical to that of IBuf indices. Apart from this a double ring buffer – or **d-DBuf** – is identical to a v-DBuf. Figure 5 shows the throughput for different types of buffer. We keep the distance between consumer and producer sufficiently large to prevent caching from skewing the results. We use different packet sizes to incorporate both the computational (per packet) and the memory-read (per byte) overhead . The compared buffers differ slightly in size, as for each we chose the bounds so that they are cheapest to calculate with (powers of 2). For example, a v-DBuf is larger than a p-DBuf, because it also contains markers. For reference, we also show allocation strategies using the `dlmalloc` general allocator ("malloc") and a buffered version of the same, which allocates 10 slots at a time ("10malloc").

The figure shows that baseline ring buffer performance is about twice as high as that of the general allocator. Although buffering malloc calls amortises costs for small blocks, it does not increase maximally obtainable results. We further see that p-DBufs indeed suffer from internal waste. For maximum sized slots p-DBuf performance is in line with that of v-DBuf and d-DBuf. As packet size is reduced, so is relative p-DBuf throughput. Up to 30% of throughput is wasted in the worst case. d-DBuf and v-DBuf results are on par. This was to be expected, as the only advantage of d-DBufs (seeking) are not evaluated in this test.

Figure 6 shows the impact of buffer size and per-call block size on throughput. We show only the v-DBUF, results are similar for other implementations. Both the buffer sizes and the read/peek sizes range from 64B to 64MB. The results for buffers smaller than 16KB coincide with that of 16KB. Again the influence of cache sizes on throughput is clearly visible.

B. Handling Resource Exhaustion

Because ringbuffers are statically allocated, they may suffer from memory exhaustion long before heap allocation fails. For this reason *Beltway buffers* must handle out-of-memory (OOM) errors themselves. In a ringbuffer an OOM condition occurs if the consumer is a whole buffer length behind the producer. Aside from standard tail-drop, *Beltway buffers* can choose from three lossy synchronisation strategies that are particularly suited to network traffic: Slow Reader Preference (SRP), Medium Reader Preference (MRP), and Fast Reader Preference (FRP). All methods are based on the idea that performance (and fairness) can be improved if occasional packet loss is acceptable. They differ in how far they deviate from tail-drop. As its name implies, MRP is a compromise

between the other two. We will first introduce the extremes and then show how MRP manages to be nearly as fast as (and scales with) FRP, but gives the stable droprate expectations of SRP.

SRP is also known as non-blocking tail-drop. It silently stops producers from accessing the ring by dropping all `write()` calls. SRP is named *slow reader* because it must recalculate the position of the slowest consumer to know whether write requests must be dropped, and the slowest reader determines the throughput. Calculation involves a sort over all read pointers. It thus scales worse than linearly (*n log n*) with the number of readers.

To circumvent the calculation we developed *FRP*. FRP blocks nor drops; the writer simply overwrites whatever is in the buffer. Consumers individually compare their location to that of the producer to calculate whether an item has been overwritten. For this to work pointers must be absolute, not 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 slow consumers are punished for their behaviour and fast consumers are no longer slowed down by tardy readers. Especially in multi-user environments it is important to shield processes from each other. A secondary advantage is that with FRP producers do not have to be aware of consumer metadata. Across the userspace/kernelspace boundary an FRP-enabled buffer is easily shared read-only. With SRP/tail-drop processes must notify in-kernel producers of their position, either through shared memory or (costly) system calls.

Unfortunately, FRP introduces complexity of its own. On a miss we need to decide how many blocks are skipped: moving the tail forward by a single slot minimises droprate in the short term, but because OOM handling is more expensive than normal processing this can result in thrashing. Because an FRP-enabled buffer is completely lock-free, data can be overwritten during a read. We place the burden of guaranteeing correct behaviour on the consumer, forcing it to check block validity both before and after accessing it. If need be, buffers can export blocks in a safe manner at the cost of an extra copy. For legacy applications this copy is free because it comes with the POSIX `read(..)` call. Real FRP-aware applications see higher throughput, because they can replace the copy with `peek` plus cheaper bounds checks. Streaming media and network monitoring applications are easily made FRP-aware. Even non-streaming applications can increasingly handle occasional data-corruption, e.g., modern block-based p2p clients.

Medium Reader Preference *(MRP)* logically sits between FRP and SRP. The model is simple and centred around a single shared read pointer R that is updated by each consumer c, which also maintains a private read pointer r_c. The producer simply treats the buffer as tail-drop with a single read pointer R. When a consumer with read pointer r_c^{orig} is scheduled it consumes n bytes of data and subsequently executes $R = r_c^{orig}$ and r_c^{orig}+=n. In other words, it updates R to its own *previous* read pointer. Assuming consumers are scheduled in round robin fashion, all other consumers now have exactly one chance to read these n bytes before c is scheduled again, at which point it will set R to $r_c^{orig} + n$. Now the n bytes may be overwritten. Functionally, MRP mimics SRP except for very tardy readers. Performance-wise it resembles FRP, because slow readers cannot hold up fast readers and synchronisation is minimal. It combines the advantages of both.

By default *Beltway buffers* employ SRP to guarantee application data integrity, but developers can manually switch to FRP or MRP. Doing so will benefit MON applications as they switch between tasks for network packets. Deep within the OS FRP already replaces SRP, because there we do not have to cater to application expectations and FRP can more effectively exploit hardware characteristics such as burst transfer.

C. Peripheral Hardware Integration

Network devices generally communicate with the host through ringbuffers, which makes integration into *Beltway* straightforward in principle. The strict separation of buffer interface and implementation further makes it feasible in practice. Technical details, such as hardware bugs, are dealt with cleanly through device-specific ring implementations. Architecture specific hints for caching (e.g., Intel's Direct Cache Access) can be embedded here as well. Upcoming header splitting, whereby transport layer headers and payload are sent to separate rings, will enable zero copy data reception. Finally, with lower per-packet stack traversal cost, *Beltway buffers* reduce the need for Large Receive Offloading (LRO) and TCP Segmentation Offloading (TSO).

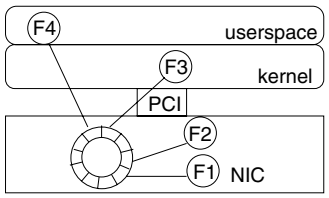

Fig. 7. Remote address spaces

In high-speed communication, peripheral media are another critical link. Special care must be taken to keep them from becoming the performance bottleneck. For example, Figure 7 shows a high-performance NIC that gives direct host access to its on-board DBuf. With *Beltway buffers*, the application or kernel interface is not different from that of host-buffers. Only latency and throughput differ, and are affected by the chosen buffer implementation. A zero-copy implementation intuitively appears optimal, but fails to make use of hardware support, such as DMA. For sequential access DMA is considerably faster, if used correctly. A practical issue concerns DMA efficiency with metadata updates. The PCI bus is a clocked bitpipe shared via bus arbitration. To maximise throughput, the bus must be used in burst mode and bus arbitration must be avoided. In other words, data must only be transferred in one direction. FRP is uniquely suited to this scenario and therefore the default synchronisation strategy across a shared bus.

VI. Practical Use In Networking

To demonstrate practical use, we engineered popular network APIs to use *Beltway buffers*: BSD sockets and the packet capture (pcap) library. Additionally, with an in-place TCP

reassembly routine we demonstrate how shared ringbuffers can bring about savings higher up in the network stack.

a) PCAP: The pcap library implements an application programming interface for network packet capture. It is used by well-known MON tools such as `tcpdump`, `nmap`, `Ethereal`, and `Snort`. We provide the same functionality by pushing all incoming packets into a DBuf and all interesting references (as selected by a standard BPF filter) into an IBuf. Pcap on top of *Beltway* has several advantages over traditional implementations. First, packet copying is minimised, not only across vertical boundaries (like kernel-userspace), but also horizontally. For instance, administrators may start up tcpdump alongside existing applications and simply share the original buffers. Second, rather than pulling each packet individually out of the kernel, we minimise context switching by accessing the DBuf directly from userspace and processing many packets in one timeslice.

b) BSD Sockets: We also implemented sockets on top of *Beltway*. In this case, too, all calls operate on locally accessible DBufs and IBufs. `recv`, `send` and related calls are protocol-specific wrappers around `read` and `write` that combine data with headers. The `accept` call works almost the same: it waits on an IBuf containing elements pointing to new flows. To wait on multiple file-descriptors, `select` and `poll` are used. Standard implementations of these have been shown to scale inefficiently [15]. To circumvent polling on multiple IBufs, we supply a separate IBuf into which all data destined for any of a process's file descriptors is written. The solution is not generic, but implements the standard use of `select`: to wait for any of all open streams to become active.

c) In-place TCP Stream Reassembly: Recreating a continuous stream of data from packets is expensive because commonly it incurs a full payload copy. TCP is especially difficult, as it allows data to partially overlap. *Beltway buffers* offers a TCP reassembly mode where streams are reassembled in-place, i.e. in *zero-copy* fashion. In the common case, when packets do not overlap and arrive in-order, our method removes the cost of copying payload completely. Instead, we incur the substantially smaller cost of bookkeeping the start and length of each TCP segment.

Our TCP reassembly design is based on the insight that consumers do not need access to the streams continuously. They only need to receive blocks in consecutive order. In in-place TCP, segments are received in DBufs. The offset pointer in the IBuf is then set to point to the start of the segment payload, rather than the start of the encapsulating packet. Executing a `read()` call results in returning an amount of data of at most one segment's payload length in size. `peek()` even returns a pointer directly into the original segment.

To quantify how indirect stream reassembly measures up to regular copy-based reassembly we compared them head-to-head. The two functions share the majority of code, only differing in segment bookkeeping. Indirect reassembly outperformed the copy-based method for all but the smallest packets (below 100B). Only in that case is accounting cost discernible.

	posix	16kB	64kB	256kB	1MB	4MB	pthreads
cswitch	155633	62996	17427	4630	1192	622	837
pgfault	646	706	743	887	1461	3760	582

TABLE I
BREAKDOWN OF PIPE OVERHEAD

VII. EVALUATION

We now compare our work head-to-head to Linux 2.6.16. All tests were executed on an AMD Sempron 3000+ with 128KB of (unified) L2 cache memory.

We begin with an IPC test. Figure 8 compares UNIX pipe throughput of Linux and *Beltway buffers*. *Beltway* does not use the `peek` optimisation, thus both systems copy the same amount of data. Any performance improvement comes from a reduction in context switching. As upper bound on achievable performance we also show a threaded application that communicates through shared memory. This application outperforms Linux pipes by a factor of 2.5. In between them are 5 SRP v-DBufs of varying size. The fastest buffer is neither the largest, nor the smallest. Initially, performance grows with buffer size as the number of context switches drops. Ultimately, however, the TLB runs out of entries and starts thrashing, which nullifies further context switch savings. Table I plots the number of context switches and page-faults incurred as a function of call size. As expected, context switch count diminishes with size. Page fault overhead only becomes pronounced when the working set exceeds TLB size (here 256kB). Indeed, peak throughput occurs with rings sized between 64 and 256KB (for 4KB pages). We observed the same result for the threaded application, but have only plotted its optimal result for brevity.

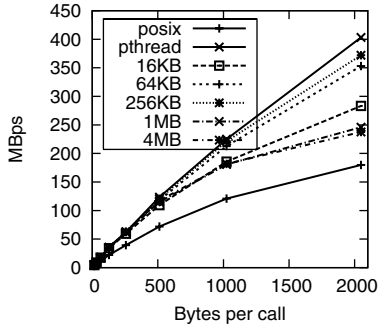

Fig. 8. Evaluation of pipes

A conservative configuration of *Beltway buffers* outperforms linux pipes. Next, we compare this configuration to one using `peek`, with and without the fast splice optimisation. Figure 9 shows sustainable throughput for an FS scenario, where data is read from the page cache and written to a network transmission *IBuf*. Highest throughput is obtained with `p/o`: the `peek` equivalent of read-only access. The rate can even exceed the physical bus limit, as no data is directly touched. This mode has no practical use, however, and is only shown as upper bound on performance. Barely slower is `fast peek/write`, which combines `peek` with splicing. This does not touch any data either, but writes out an index. `read` cost becomes apparent when comparing these results with the other two, `read/only` and fast `read/write`. They are an order of magnitude slower. Worst results are obtained when we cannot use splicing, but instead must write out data:

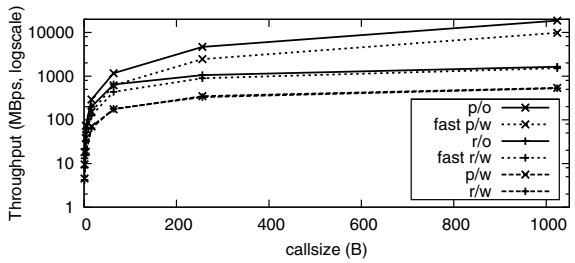

Fig. 9. Read and write performance

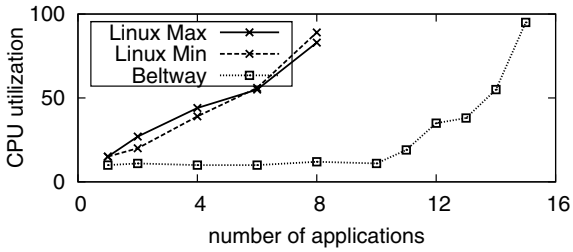

Fig. 10. Tcpdump scalability

throughput drops again, to a third. We observe that writing is the main bottleneck from the observation that `peek/write` and `read/write` are equally fast, while we see in other cases that `peek` clearly outperforms `read`.

Figure 10 compares cost, in terms of CPU utilisation, of Beltway `tcpdump` to two Linux configurations: with minimal and maximal capture length (68 and 1500 bytes per packet). To investigate scalability we process only a moderate datastream: 100 Mbps of maximum sized packets (i.e., 75000 pps). The figure shows that our version is about 25% faster for a single application. More interesting savings occur when we run applications in parallel. Whereas standard tcpdump scales linearly with the number of packets, our version incurs no significant overhead for up to 9 applications. With 10 applications, however, the system starts thrashing. By inspecting the number of voluntary and forced context switches independently we learnt that, although involuntary switching increases with each application, thrashing does not occur until the number of *voluntary* switches starts *de*creasing. That is a sign that applications cannot process all data during their slot, which starts a snowball effect: thrashing.

Although we expected standard tcpdump to be memory bound, Figure 10 shows that the minimal and maximal capture length versions have roughly the same overhead. Thus, tcpdump is not directly memory bound. Indeed, 100 Mbit of data may be copied tens of times before a modern memory bus is saturated. The real bottleneck is that tcpdump switches for each packet. Standard tcpdump switches 19 times as much as a single *Beltway buffers* tcpdump instance (77000 vs 4400). Even for 10 parallel applications, our version switches only a 5th the amount of a single standard tcpdump (16000).

In short, we can monitor traffic in parallel with other applications with minimal performance degradation. More interestingly, as switching costs seem to dominate overhead, the same advantages exist when accessing non-overlapping data.

Finally, we applied *Beltway buffers* in an intrusion prevention system (IPS) embedded in a programmable network card based on the Intel IXP2400 network processor. The IPS applied in-place TCP reassembly, full payload regular expression scanning and layer-7 protocol validation. Despite the computational overhead, the system achieved a throughput of almost 1 Gbps in the worst case. Interested readers are referred to [2].

VIII. Conclusions

Beltway buffers is a buffer management system that reduces I/O overhead, in particular for network applications. *Beltway buffers* consigns all data to shared ring buffers to amortise allocation and data movement overhead. Efficient access to these buffers additionally reduces context switching and cache invalidation. The *Beltway buffer* system provides a familiar POSIX file API to its users, which makes it backwards compatible with existing Unix-like operating systems.

Acknowledgements

We would like to thank Peter Druschel, Philip Homburg, Kees Verstoep and Tomas Hruby for commenting on earlier versions of this paper.

References

[1] "splice(2) - linux man page," http://linux.die.net/man/2/splice, 2006.
[2] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos, "Safecard: a gigabit ips on the network card," in *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.
[3] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on i/o performance," in *Operating Systems Design and Implementation*, 1996, pp. 277–291.
[4] P. Druschel, M. B. Abbott, M. A. Pagals, and L. L. Peterson, "Network subsystems design," *IEEE Network*, vol. 7, no. 4, pp. 8–17, 1993.
[5] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Symposium on Operating Systems Principles*, 1993, pp. 189–202.
[6] J. Pasquale, E. W. Anderson, and K. Muller, "Container shipping: Operating system support for i/o-intensive applications," *IEEE Computer*, vol. 27, no. 3, pp. 84–93, 1994. [Online]. Available: citeseer.ist.psu.edu/pasquale94container.html
[7] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Io-lite: a unified i/o buffering and caching system," *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 37–66, 2000.
[8] R. Govindan and D. P. Anderson, "Scheduling and ipc mechanisms for continuous media," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 1991, pp. 68–80.
[9] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: a user-level network interface for parallel and distributed computing," in *Proceedings of SOSP15*, 1995.
[10] I. Pratt and K. Fraser, "Arsenic: A user-accessible gigabit ethernet interface," in *INFOCOM*, 2001, pp. 67–76. [Online]. Available: citeseer.ist.psu.edu/pratt01arsenic.html
[11] V. Jacobson and B. Felderman, "A modest proposal to help speed up & scale up the linux networking stack," http://www.linux.org.au/conf/2006/abstract8204.html?id=382, 2006.
[12] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson, "Design principles for accurate passive measurement," in *Proceedings of PAM*, Hamilton, New Zealand, Apr. 2000.
[13] L. McVoy, "The splice I/O model," www.bitmover.com/lm/papers/splice.ps, 1998.
[14] K. Claffy, G. J. Miller, and K. Thompson, "The nature of the beast: recent traffic measurements [...]," in *Proc. of INET'98*, 1998.
[15] A. Chandra and D. Mosberger, "Scalability of linux Event-Dispatch mechanisms," in *Proceedings of USENIX 2001*, 2001, pp. 231–244.