Technical Report IR-CS-028

Using Beltway Buffers for efficient and structured I/O

September 2006

Willem de Bruijn Herbert Bos

Department of Computer Science, Vrije Universiteit Amsterdam

{wdb, herbertb}@few.vu.nl

abstract

We describe *Beltway buffers*, a new buffer design for flexible, high-performance communication paths in commodity operating systems. We target network processing, but also other I/O (e.g., inter process communication and disk access). Advantages of the buffer system are (1) simplicity and familiarity due to a POSIX-based API that is used throughout the system, (2) aggressive copy-avoidance, (3) support for hardware accelerators, (4) judicious use of the data cache, and (5) support for discrete datablocks (e.g., network packets or video frames) as well as bytestreams.

1. INTRODUCTION

In this paper we discuss buffer management for OS-level communication paths with an eye on efficiency and flexibility. While the topic has been extensively studied in the past, we argue that unresolved issues remain and propose a new buffering scheme, known as *Beltway buffers*, to address them. The system is designed for all local communication and therefore provides all well-known primitives: sockets, file descriptors and pipes. In addition, it supports many advanced features (e.g., user-visible disk caches, multiway named pipes, copy avoidance, prefetching, interrupt mitigation, and splicing [19]). Internally, *Beltway buffers* are structured quite differently from existing systems and offer many attractive properties that provide both performance and structure.

Operating systems at the network edge constitute the main bottleneck in networking. The problem has dogged OSs for decades [7, 13,16,18,23], and given the fairly constant trends in network speed, memory latency, peripheral buses, and processing power, we expect it to grow worse rather than better in the future. Efficient communication paths are therefore crucial for performance.

Common OSs combine archaic notions of networking with ad-hoc fixes. The ensured patchwork harms performance through unnecessary context switching and data copying. For instance, simple tasks such as reading a file from disk and writing it unmodified to the network (e.g., by a web server) incur superfluous context switching and copying to a user process that immediately copies the data back to the kernel for transmission. On x86 architectures, context switching requires expensive updates to the global and local descriptor tables. The two redundant copies are even more expensive and indirectly muddle the TLB with serious consequences. Ad-hoc fixes, like adding a new `sendfile()` system call to the kernel, are used to circumvent this problem, but do not work in the general case. In fact, here it does not even work in the opposite direction.

While several projects have aimed at developing stacks in a structured, modular fashion, early attempts (e.g., STREAMS [26]) incurred significant performance penalties. Nevertheless, the concepts proved durable as witnessed by more recent work such as Scout [21] and Click [20]. Druschel proposed FBufs [13] to reduce the overhead. FBufs were reused in IO-Lite [23]. Both projects gear for speed and reuse memory mappings. However, interfaces are unusual and complex, which makes it harder to handle new stacks and hardware. In essence, the interface affects portability of applications, as IO-Lite requires changes to both applications and network stacks, and hardware requires early demultiplexing capability. Also, to reuse VM mappings, communication paths must be fairly static. In addition, by handling and mapping datablocks individually, signal coalescing (known in networking as interrupt mitigation) is difficult as each block incurs a ‘signal’.

This paper focuses primarily on *buffering* rather than *processing* (e.g., protocol stacks), but adding more structure to communication paths is an explicit goal for us also. Processing will be mentioned in passing, but is largely orthogonal.

In summary, then, buffering in existing OSs is either convoluted or slow. Indeed, clean abstractions and modular design appear at odds with the heuristic that Integrated Layer Processing (ILP) improves performance [8]. Part of the motivation for our work is the conviction that structured approaches and modular design are still preferable to less structured ones and that careful design will alleviate potential bottlenecks. We believe that in the long term modular approaches win, because they evolve more easily.

In addition, convoluted systems make it harder to integrate new hardware in a clean manner. For instance, to support high link rates many devices offer on-board processing of network traffic. Examples include TCP checksumming and segmentation offloading, port-based early demultiplexing, pre-defined counting and filtering functions (Juniper routers, DAG cards), and even full programmability of the network card (network processor boards). To integrate such hardware, one is often forced to tear apart the network stack in the OS kernel, or to make the device look like a plain old NIC. The problem is urgent: OSs are hardly able to keep up with link rates today, and the problem is expected to get worse. Be it on-chip (similar to current SSE/MMX extensions), as a plugin board, a network device connected to the host, special purpose hardware will be increasingly present in future architectures. Existing buffering systems were never designed for such complex configurations. They not only fail to integrate the devices, but also overlook communication channel heterogeneity (e.g., on-chip, PCI, PCI-Express).

In this paper, we propose and evaluate a new and structured buffer implementation for high-speed communication, known as *Beltway buffers.* The system explores an extreme in the design space for communication paths. Where most existing systems are careful not to waste memory, we sacrifice memory to gain performance, structure and flexibility. *Beltway buffers* feature aggressive copy avoidance, judicious use of the cache, and support for both continuous streams and discrete data blocks. At the same time, they are simple to use and easy to extend, as all complexity is shielded behind well-known APIs that are used throughout the system.

Summarising the specific contributions of this paper, our first contribution is that we include all processing *spaces* (applications, the kernel, programmable dedicated hardware, etc). In contrast to other solutions, we do not assume a single address space. Heterogeneous address spaces are supported simultaneously (e.g., different 'smart' NICs, programmable routers, etc.), while presenting a single coherent system view to the applications. Our second contribution is that our shared ring buffers provide a datapath without dynamic allocation or VM mapping, while maintaining security at fine granularity. Our third contribution is that we developed a highly structured buffer system without sacrificing performance.

Two applications, denoted by **FS** and **MON**, will be used as running examples throughout this paper. FS represents a file/web server pushing data from disk to the network. MON is a monitor that sniffs on network traffic (e.g., an intrusion detection system that reconstructs streams and looks for patterns in the traffic). While these applications do not cover all possible use cases of *Beltway buffers*, they help illustrate many of their features and advantages.

Unlike existing solutions, we eschew per-block (or per-packet) buffering, i.e., all solutions that are similar to Linux `sk_buff`s or BSD `mbuf`s in spirit, if not in implementation. In per-block buffering, a pool of buffers is maintained and whenever a packet arrives, a buffer is allocated. Pointer queues and reference counts are maintained to reduce copying and maximize sharing.

Beltway reduces allocation overhead by moving all buffering into shared rings. So all incoming MON packets and outgoing FS packets are written directly in a pre-allocated buffer at the current write position of a receive and transmit buffer. VM operations like mapping buffers to user space applications are also executed statically and mappings remain in place until applications remove them explicitly. Similar motivations led to Deri's ncap/`PF_RING` [11] and our own FFPF [2]. While `PF_RING` is a fairly modest attempt to improve the speed of a single socket in the current Linux network stack, it demonstrated the efficiency gain of remapped ring buffers.

Our buffers have uniform and familiar (POSIX-based) interfaces that are used throughout the system. Unlike existing work we adapt buffer implementations to the applications and underlying hardware. For instance, while the interfaces are constant for ease of use, specialised buffer implementations handle transfers across the PCI bus, so packets for MON programs are DMAed efficiently to host memory. In addition, buffers can be shared between arbitrary groups of spaces, but access policies are handled on a per-buffer, per-space basis, i.e., control is fine grained. In particular, we define *security groups* of applications permitted to share buffers. Read and write access to buffers is controlled in a POSIX file manner, by specifying access rules for user, group, and other.

Through system-wide sharing, *Beltway* removes the need to cross protection domains for each block. It is geared for high-performance through copy-avoidance, context-switch mitigation and optimisation of cache-usage. Even so, as a single copy is sometimes cheaper than no copy at all [2], we do not aim for zero-copy *per se*. Rather, we strive to optimise the copying scheme to provide highest performance.

Beltway caters to packets as well as streams such as TCP flows, and allows for straightforward integration of complex, heterogeneous hardware. The buffer system is able to cross boundaries to access all resources, from the application-level down to the network card or even remote machines. While our examples often concern high-speed networking scenarios, as these are perhaps the most demanding, the *Beltway buffers* are intended for communication paths in general. For instance, we use them for disk I/O and named pipes also (e.g., common pipes are sped up by a factor of two; sending data from disk to a transmit buffer by an order of magnitude).

Development of the *Beltway buffers* started in 2004 and draws heavily on our experience with fast packet filtering in FFPF [2]). FFPF was designed for efficiency, but limited to packets and MON applications. We redesigned the buffer system entirely and approached the problem by implementing and evaluating IO and communication mechanisms at different levels independently, before combining them in a system wide buffer system. Taking such a bottom-up approach helped us test assumptions and address many practical issues that a top-down solution might have missed. 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, Wikipedia is about to use *Beltway buffers* to monitor user requests for pages. We also used them to implement practical high-speed intrusion prevention on embedded hardware [10].

While working on the Wikipedia MON application, we also observed a growing need for multiple applications accessing the same data. For instance, an IDS scans incoming traffic, while tracers monitor progress, and loggers store data from different applications on disk. In current systems, doing so incurs copies for each application and is often prohibitively costly. In *Beltway*, copying and context switching is minimised and multiple access to the same data is cheap. This is especially useful for MON applications.

The remainder of this paper is organised as follows. In Section 2, we consider related work. Section 3 discusses the *Beltway* design, while Section 4 shows how buffer implementations vary for optimal performance. Section 5 discusses how the buffers are used to provide abstractions like sockets, pcap, etc. In Section 6 we evaluate our work and in Section 7 we draw conclusions.

2. RELATED WORK

STREAMS is the earliest attempt at a structured network stack in which modules are connected through full-duplex message queues [26] . Extensive queueing reduced performance compared to sockets. The x-Kernel adds session handling, and processing extends across protection domains, a feature added in anticipation of microkernel uptake [16]. Scout improves these concepts by extending the processing stack into application processing [21]. The system needs early demultiplexing to ensure high-performance. Scout has been used in conjunction with a programmable NIC to build an extensible layered router, Vera [18]. Like Click [5], Vera only works on packets. Each execution layer's implementation is complex and handcrafted. Extensibility does not reach the dataplane layer. Click paths are static and no per-session or per-flow paths are needed, as Click is limited to layers three and lower.

Application processing was targeted by SEDA [27]: a network server based around the dataflow model. SEDA is inherently scalable through its use of modules interconnected by queues. Through Random Early Dropping (RED) [14] it also increases server robustness. SEDA is based on costly queueing and scheduling and no process separation or security concerns are taken into account.

Dataflow-like processing as advocated by these projects appears to be at odds with [8] which states that ILP improves performance at least for simple functions [1]. This is certainly true when dataflows are implemented through explicit scheduling and queueing. Procedure calling and pass-by-reference reduce the impact. As explained in Section 5.3, we transparently remove calls to overcome the problem, thus combining the performance of ILP with the modularity of dataflow processing.

Copy-avoidance mechanisms replace copying with virtual memory (VM) techniques such as page remapping and copy-on-write. Common solutions work at block granularity. Blocklevel remapping trivially ensures security, but at at cost: recurrent modifications to VM structures reduce efficiency. Brustolino [3] categorised previous efforts and showed them to perform roughly identical. Druschel *et al.* describe copy avoidance ideas for network buffers [12] and subsequently translate these into Fbufs [13]: copy-free communications paths across protection domains that sometimes remove the per-block costs. Paths are efficient if mappings can be reused between blocks, for which early demultiplexing is required. This is also true for container shipping [25]. Fbufs are later incorporated in IO-Lite [23], a transport system that integrates all buffering in the OS. IO-Lite introduces a composite datatype of pointers to immutable buffers to replace copying and copy-on-write mechanisms throughout the OS. Interfaces are new and somewhat complex.

As mentioned earlier, *Beltway buffers* move away from per-block allocations and amortises allocation overhead by moving all blocks into shared ring buffers. Furthermore, while buffers have a uniform interface, implementations are tuned for performance. Between protection domains arbitrary groups of buffers can be shared; access policies are handled on a per-buffer per-space basis to ensure fine-grained control. Signalling is mitigated and *Beltway*'s index buffers are comparable to `rtsignal` queues, while wakeup strategies are implemented on a per-boundary basis as optimal choices vary across hardware (polling, interrupts, timer-based).

Govindan *et al.* [15] introduced memory-mapped ring buffers to reduce cross-space communication. Rings are attractive for single producer, multiple consumer type sharing because they are lock-free. This benefit comes at a cost: ring buffers pre-allocate space, wasting memory. As memory size grows faster than memory-bus performance this trade-off is increasingly viable. Govindan uses buffers as one-to-one pipes between an application and the kernel. *Beltway buffers* extend this basic notion by allowing shared buffers between arbitrary spaces, regardless of whether they are applications, OS subsystems, or embedded logic. Furthermore, *Beltway buffers* implement sharing widely throughout the system: arbitrary sets of buffers can be shared between arbitrary sets of spaces.The resultant buffer subsystem is generic. It underlies networking, but can be used for any cross-space communication system.

3. ARCHITECTURE

For memory and I/O bound tasks, minimising transport overhead is crucial. Copying must be avoided where possible, caching must be exploited aggressively and runtime allocation must be averted. The *Beltway buffers* design revolves around moving all data transport throughout the processing stack (process, kernel, peripheral devices) into shared ring buffers. By separating the functional components (e.g., the protocol stack or other processing steps) from the transport aspects (the way data is buffered and moved across space boundaries), we reduce clutter and allow for additional optimisation. This paper focuses on the transport plane. Functional aspects will be summarised in Section 5.

3.1 Multiple Rings

A trivial solution to achieve the above goals for reception is to make drivers receive all data in a single, shared ring buffer that is contiguous in memory, mapped to the applications, and fits a set of data cache lines. Such a ring buffer has desirable cache behaviour and incurs no copies whatsoever. In addition, runtime allocation is non-existent. This simple solution was originally used with minor modifications by projects like ncap/`PF_RING` and our own FFPF [2]. However, a single ring is too simplistic a solution to replace all communication paths, for the following reasons.

Firstly, unacceptable security concerns arise when there is no control over who has read and write access to which packets. For example, the MON-like applications of normal users should not be able to access other users' data. A shared buffer may be fast in principle, but runtime access coordination is needed to ensure that users do not tamper with each other's data. Indeed, security concerns were raised against solutions that work on shared buffers at coarser granularity than packets by the authors of FBufs [13].

Secondly, in the presence of multiple processors or cores, coupling a buffer to a single core boosts performance, primarily because it keeps the data in the same cache. In contrast, a single ring is shared by all processors with serious consequences for performance. What we want to avoid is silly cache behaviour, whereby an update by consumer 1 leads to a cache invalidation for consumer 2 even if there is no data dependency. For instance, when consumer 1 updates a read pointer, or updates data that is not of interest to consumer 2, there should be no cache invalidation. The same is true for metadata updates by producer and consumer. Similar objectives led to Van Jacobson's proposal of introducing 'netchannels' to Linux [17] whereby the processing of a packet, including TCP, is tied a single processor. The *Beltway buffers* discussed in this paper should facilitate such behaviour, while also offering other benefits.

Thirdly, besides catering to multiple cores performing end processing on the data, we must also attend to multiple execution spaces at lower levels. For instance, network cards may run some functionality on the card (filters for MON, say, or checksumming for FS) either using on-board buffers [10], or by managing their own buffers in host memory [9]. In the former case, the problem is that of distributed memory. In the latter, the memory is physically co-located, but at different addresses. In either case, we need to reconcile the buffers to provide a single coherent view on the data.

Finally, transformation of data may be hard to do in-place because of security or buffer space limitations. In that case, data rewriting may be better done in a different buffer. Well-known examples of data transformation include common TCP reassembly, encryption, forward error correction and video encoding/decoding.

3.2 Advantages of Rings

Most modern OSs use dynamic buffering per data block. A (possibly cached) pool of buffers is maintained and whenever data arrives (e.g., a packet arrives in MON, or disk data in FS applications) an appropriate buffer is allocated and associated with the data. Pointer queues are maintained to reduce copying and maximise sharing. The need for a buffering system beyond pointer queues and `malloc/free`-style dynamic allocation is motivated by the overhead of runtime allocation, the cumbersome per-page VMM trickery, and the limited use of pointer queues.

Runtime allocation. At high rates, we cannot afford to allocate memory at runtime. In contrast, ring buffers are allocated only once. We trade optimal memory utilisation for performance.

Runtime page remapping. If multiple protection domains share data elements, current solutions use cumbersome per-page memory mapping at runtime which is slow unless it can be reused. As a result, a MON-like intrusion detection system (IDS) that receives a copy of all traffic incurs overhead for every packet. Instead, in *Beltway buffers*, entire rings are shared permanently.

Rich pointers. Pointer queues improve sharing, but pointers are limited in practise, because they are not valid across address spaces. Furthermore, a pointer provides no information about the data's context (e.g., the TCP flow, or the movie frame). For this reason they are often embedded in heavyweight datastructures (e.g., `sk_buffs`), which introduce locking that is otherwise unnecessary. Rich pointers pose an alternative. In *Beltway*, each consumer has a private set of pointers to data blocks, known as IBufs, that are valid across all spaces. As a result, it has a private view on the data. The pointer may convey meta-information about the data, such as the result of prior classification (e.g., the TCP flow) while minimising structure locking. The meta-data frequently determines the next processing step without needing to read the actual data block. Moreover, compared to datablocks, IBufs are small and can often be served from the cache. We discuss IBufs in detail in Section 3.4.

3.3 Allocation and Memory Mapping

Beltway buffers minimise copying overhead through aggressive sharing. In most cases, a block is copied into a buffer once and accessed by everyone from there. For instance, FS applications like webservers read files in a cache from where they are pushed to the transmitter directly and combined with headers, scatter-gather style. MON applications receive data in a buffer and provide all processing functions with shared access to this buffer. *Beltway* minimises copying both *vertically* (e.g., between NIC, kernel and applications for, say, FS sending data to the network), and *horizontally* (e.g., when demultiplexing the same data to multiple applications, for, say a MON application sniffing all incoming traffic). As calls to the general memory allocator are expensive, ring buffers are allocated in advance. The buffer is allocated as a contiguous virtual memory block, e.g., during application initialisation. Because the entire buffer is a single block of data, it can be mapped into remote spaces in a single operation.

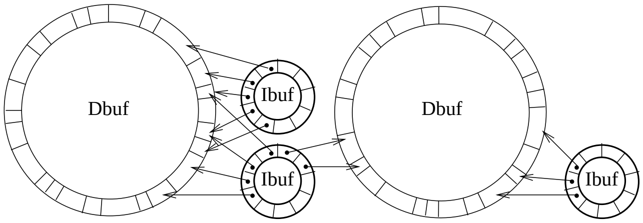


Figure 1: Chain of rings with data and index buffers

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

A practical allocation detail is whether to use physical or virtual memory. The overhead of virtual memory is not strictly necessary. However, as physical memory is difficult to reserve at runtime and micro-benchmarks showed no significant impact, our rings are allocated from physically contiguous virtual memory as much as possible. We strive to allocate the memory areas such that they occupy minimal space in the TLB, e.g., by a single large page mapping instead of multiple smaller ones. If no such memory is available the allocator silently resorts to non-contiguous mappings.

Either way, remapping the buffer in a single operation decreases runtime overhead. The operations performed for setting up shared memory (as carried out by the POSIX `mmap` call) are expensive. Per-page copy avoidance mechanisms incur this cost for each block. Even though the benefits outweigh these costs, profits are not maximised. In contrast, in a shared buffer, mapping costs are amortised over all blocks and functions, rendering them irrelevant.

We prefer to remap shared buffers among spaces. For instance, a userspace MON application has direct access to memory mapped kernel buffers, and FS uses a page cache directly visible at user level. However, shared memory is not always feasible. Across constrained buses (e.g., the PCI bus) maintaining a shared memory system has too great an impact on performance. Over such communication channels, forwarding is implemented as a pipe, because copying the data once minimises communication. Furthermore, transfer methods can be optimised to match the characteristics of the underlying medium. In Section 4.4 we introduce efficient strategies for crossing constrained resources.

3.4 Data Buffers and Index Buffers

At the heart of the *Beltway buffers* are collections of ring buffers of varying sizes and use-cases (Figure 1). Data is stored in a shared data buffer (DBuf), while pointers to a subset of the data are maintained in much smaller private index buffers (IBuf).

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.

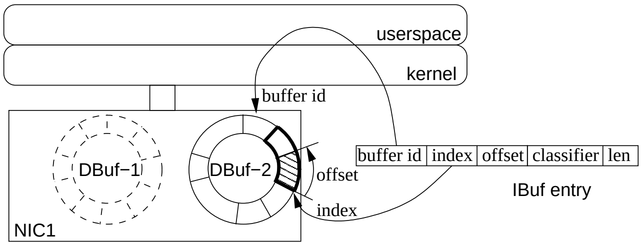


Figure 2: IBuf entry points to a byte in a specific DBuf on NIC$_1$

Different IBufs may point to the same (or different) blocks in the same DBuf. In other words, consumer DBuf access is non-exclusive. In contrast, to avoid locking we allow only a single producer per DBuf. Non-exclusiveness works both ways: IBufs can freely point to blocks in different DBufs. This is important when a consumer is interested in data arriving over multiple paths, e.g., a MON application monitoring two NICs.

Another example is the processing path of an SMP machine. In the presence of multiple cores or processors, each consumer may run on a separate core. By maintaining the data list for each consumer in a separate IBuf, interference between consumers 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, consumers process disjoint sets of data blocks. For instance, an application processes its own set of IP packets or frames. By aligning buffer slots on cacheline boundaries, we fully decouple data access on multiple cores.

We also use IBufs for security restrictions. At this point we only sketch the idea, as security and buffer access control are discussed in Section 3.8. When restrictions do not apply (e.g., for MON-like sniffers with root access), IBufs help to transparently point to multiple rings. However, ordinary users may perhaps only sniff, and hence point to, their own DBufs (or those of their security group). Neither scenario incurs additional copies, as all references are to the same DBufs. Similarly, on the transmission path IBufs help converge a userspace ring containing payload and a kernel ring containing packet headers. Scatter-gather drivers can trivially read IBuf indices to prepare PCI-DMA communication.

3.5 Indirection Details

As DBufs may reside in different spaces and the *Beltway* system does not assume a single address space, IBuf elements have to be valid across all 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 in that buffer to select a block of data, and an offset in the block. Prior to accessing, say, a network packet for the first time, the IBuf pointer is translated to a local pointer which is cached for subsequent accesses. An example of an 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 5.4 we discuss in-place TCP reassembly for MON along these lines.

Although an index is far smaller than the average data block size, writing and reading still comes at a cost. Fortunately most IBufs can be optimised away. An IBuf identifies a 'stream' of data (as a subset of other streams). The actual buffer is only needed during asynchronous communication; during function calling an index can be passed directly. So when two adjacent layers in a protocol stack are in the same space, there is no need to maintain IBufs. Instead, the layers pass pointers directly. Again, the *Beltway* system handles this transparently. We identify two occasions where IBufs are needed: forwarding across spaces and exporting data to end-users. Only in these situations the *Beltway buffer* system sets up IBufs. Most applications are implemented using only one DBuf and one IBuf, regardless of the number of kernel tasks that process the data.

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 spaces. We do not presuppose a single address space operating system, as that would limit applicability. It is also not necessary, because overhead is minimal: caching pointer calculation amortises overhead over all applied functions within the same space.

When reading from or writing to a DBuf, calls directly access the local data ring. When a call is made to an IBuf, however, we do not return the IBuf contents. Instead, *Beltway* silently resolves the corresponding DBuf and return a block from there (provided the pointer stored in the IBuf is still valid). Transparent indirection makes application programming simpler, because it gives applications a private view on data without needing a separate interface.

In a shared-memory system transparent access demands an additional step. The three-level pointer in an IBuf element has a member pointing to a DBuf. If this buffer is not accessible in the current execution/address space it is silently mapped in, in a manner reminiscent of page-fault handling. When the reflection routine notices that a buffer is not accessible from the current space, it asks neighbouring spaces whether they have it. If so, the buffer is mapped in automatically. The DBuf is only mapped in when a page-fault occurs. (although the performance hit will be incurred only once). Preloading all DBufs that are referenced in an IBuf should reduce runtime overhead, but is considered future work.

3.6 A Uniform Interface

No single best ring buffer implementation for all applications exists. Some spaces allow for efficient memory sharing, while others leverage hardware support for pushing data across the PCI bus. Certain communication paths may benefit from traditional slots, while others access continuous byte-streams. Moreover, we should account for difference in hardware alignment and endianness. Dealing with such buffer peculiarities usually leads to clutter in the buffering dataplane, the opposite of a structured OS. In contrast, *Beltway buffers* hide these implementation details behind a common interface. All buffers look alike to the user. 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 access directly *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: BSD Sockets to support unmodified networking applications, UNIX pipes, the packet capture (PCAP) interface for MON applications (like `tcpdump`, `ntop` and `snort`), and an experimental composable network stack called Streamline. We will discuss these interfaces and their use of buffering in Section 5.

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 [19]. `Splice` controls a kernel buffer from userspace. It allows data to be moved to/from the buffer from/to arbitrary file descriptors[1]. FS greatly benefits from `splice` as disk data is no longer copied to userspace. MON applications use splice to save interesting data to disk with the same benefits.

The POSIX API provides familiarity to users of UNIX and other POSIX-compliant OSs. We extended its use to make the interface available throughout the system rather than only in the userspace-kernel ABI. Still, a *Beltway buffer* is opened in familiar style using:

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

The name argument is optional and can be used for shared buffers (similar to shared pipes). The fourth, also optional, parameter is an extension to POSIX. It is used to set the 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 [15]. This is reflected in the possibility of repeated calls for the same data failing in streams.

A known problem of the POSIX API is that it uses copy semantics, that is, `read` and `write` create private copies of blocks for the caller. Such semantics are safe, but also wasteful, as they must be implemented using copying or VMM modifications such as copy-on-write schemes. To circumvent these costs we extend the API with `peek`, a `read`-like function that uses weak move semantics in the nomenclature of Brustoloni and Steenkiste [3]. We prefer to call the scheme *system shared*, because there is no move action implicit, as they expect. With system shared semantics, an actor can read blocks, but because the data is shared, it is not allowed to modify them. 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);*

Instead of copying data, the subsystem can now return a pointer to the original location (somewhere in a DBuf). Note that legacy POSIX code not using the `peek` call would work unmodified, albeit sometimes less efficiently than equivalent code using `peek`. Between processes the new semantics are easily enforced by sharing the pages underlying a buffer as read-only. A major departure from the traditional POSIX API lies in access policy enforcement. *Beltway buffer* access is handled in the context of the caller, whereas traditionally calls must cross the border to kernelspace to guarantee correctness. Making the API orthogonal to the user/kernel ABI saves us many context-switches at runtime.

3.7 Buffer Structure

Internally, each buffer is implemented using three datastructures. `buf`, a per-actor structure, holds the actor's offset into the stream and private flags. `buf_data` is accessible by all actors and holds shared flags, such as access policy, reference count and most importantly the shared offset (highest written element) and buffer size. Offset and size together put a bound on the stream window. Finally, pointers to the buffer implementation's functions are reached through `buf_impl`. As *Beltway* supports specialised implementations the pointers cannot be hardwired in the control path.

For cross-space sharing only `buf_data` and the actual ring have to be remapped, which is why they are allocated together. Each space has its own set of buffer implementations, and each actor keeps its `buf` structure private. As pointers are generally invalid across spaces, some of their members do not even make sense when remapped (e.g., the function handlers). More importantly, because access patterns and costs may differ, spaces may use different logic (and thus `buf_impl` structures) for the same buffer semantics.

3.8 Security

Each buffer has a separate authorisation policy with regards to mapping, and thus access to data can be controlled in a fine-grained manner. Protection is enforced per buffer, per space. Like files, we can restrict access to a specific UNIX group (gid) or user (uid). Additionally, buffers can be shared based on process (pid).

The presented mechanism enables us to create security groups of applications that share data in common buffers. In addition, it allows administrators to install MON software (e.g., for intrusion detection or debug tracing) with the appropriate privileges which do not incur even a single additional copy.

3.9 Disk access

Beltway buffers integrate what is known as the page cache (the cache for disk data managed by the OS). One of our goals was to provide userspace processes with access to the cache, to allow applications to read from the cache without context switching. FS directly benefits from such functionality. It also enables transparent splicing, making the benefits available behind a POSIX interface.

For this behaviour 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. If no cached copy exists, an `open` call on a file by default creates a cached copy. The cached copy prefetches data from disk and serves requests for data from memory. 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. A disk cache is different from the buffers discussed so far, because – as files – it must be randomly addressable.

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

[1] Linus Torvalds is developing splice for future versions of Linux.

sendfile in many UNIX OSes. *Beltway buffers* have a more general solution that works between any two rings. By enabling what we call the **fast splice optimisation**, the two connected operations lead to splice-like functionality under the hood without requiring a change in interface. With peek no data is 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 by definition incurs one copy, the write copy and context switch are saved. As a result, even unmodified legacy applications with no notion of peek experience significant speed-ups. This cannot be made default behaviour, because it leads to data corruption if blocks are modified within the process. For FS applications like webservers it will be turned on, while for decoders it will be off.

4. SPECIALISATION

So far we have seen two types of ringbuffers: DBufs and IBufs. Their implementations differ in that DBufs hold variable sized blocks (such as network packets), while IBufs hold elements of a relatively small, fixed size. The underlying buffers are not special, however; we emphasise that all buffers export the same interface. This uniformity makes the *Beltway* system conceptually simple.

We now show, however, how modifying the buffer *implementation* benefits performance, drop rate and other characteristics. We will show multiple, orthogonal optimisations. Where applicable we note our default selection. However, if the optimum is context-dependent, we do not. A specific buffer implementation can be selected by setting the flags in the open call.

4.1 Continuous and Slotted Buffers

We can partition the buffer space in two categories: continuous and slotted buffers. In principle, continuous buffers are just slotted buffers with a slotsize of one byte. However, creating a separate implementation benefits performance. The semantic difference is slim: a continuous buffer can access data of arbitrary length. It will wrap a large block across the buffer bounds, storing some data at the end and some at the start of the underlying memory area. This is not allowed in a slotted buffer. Access costs, which are not trivial, are lower for slotted buffers because their integrity checks are simpler. For example, there is no need to implement the wrapping.

The majority of costs related to accessing data in a ringbuffer are due to checking of error conditions: is an index valid? will a block fit in the memory area? Such tests frequently involve modulo operations using the size and offset metadata. Calculations can be sped up in slotted buffers by choosing the slotsize as a power of two and replacing costly modulo checks by bitwise & operators, which are 5 to 10 times cheaper (on an x86, similar for other architectures).

DBufs may thus differ in basic memory layout (see Figure 3). For example, p-DBufs are DBufs for discrete data blocks (usually network packets) and contain fixed-sized slots large enough to hold the fraction of the packet that might be of interest to its consumers. The slot size may be the maximum packet length, the first 64 bytes, or some other value. Consumers may use the offset field of an IBuf to point to data and skip headers.

Not all data, however, can be broken into clearly distinguishable blocks. Pipes and reconstructed TCP streams do not contain natural borders, but are accessible at bytesize. Such data is kept in continuous buffers, **c-DBufs**. Should streams be multiplexed on a single c-DBuf, they cannot be told apart. A c-DBuf demands an external IBuf to break up data into segments. This can be done in one of two ways: (a) different IBufs for different streams, or (b) one IBuf with different classification results for different streams.

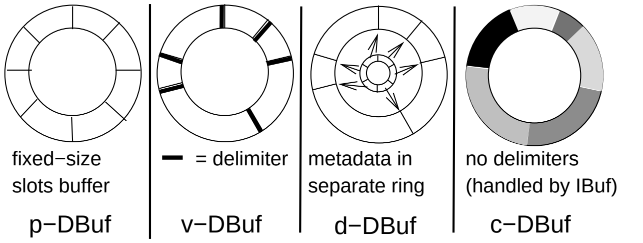


Figure 3: Different implementations of DBufs

4.2 Increasing Memory Bus Throughput

Because the memory bus is one of the bottlenecks in stream processing [24], we leverage support 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 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. Focus must lie on reducing wasteful allocations in the datapath. As discussed, ring buffers have nice properties, but they invariably waste memory. In this section, we discuss way for improving *Beltway*'s cache behaviour. We discern two types: external and internal waste. Only waste that influences performance needs to be minimised. Most of the waste incurred by shared rings is due to statically allocated, unused memory, which has no impact on cache utilisation and thus on performance.

4.2.1 On-demand Resizing

External waste is the amount of unused space between the read and write pointer. Memory locality can be increased by reducing the gap of unused slots. On the other hand, a buffer tuned too well to the average case will overflow during bursts. How to choose the trade-off between memory locality and overflow is hard to determine in general, but clearly some inefficiency is unavoidable.

External waste can be limited by adapting the buffer size. To investigate this idea, we built a resizable buffer. Resizing is similar to rehashing: when resizing is requested, a new memory area is allocated. As long as there are references to the old area both must be probed. Thereafter the old area is released. Computational overhead is minimal, but memory pressure is considerable during resizing. Because buffer sizes are based on powers of two, 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. This implementation never changes 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 in a non-resizable buffer. Both resizing implementations trade off computa-

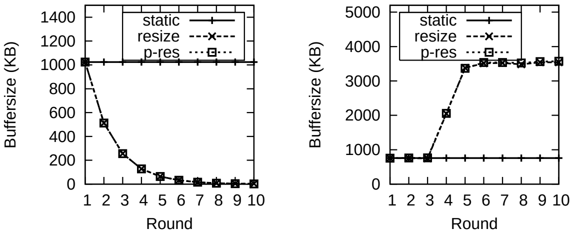


Figure 4: Resizing buffers for different rounds of resizing

tional complexity for an increase in locality of reference. We compared the two to their non-resizing counterpart in Figure 4. The plot on the right shows throughput of static buffers, resizing and pseudo resizing buffers in consecutive rounds of resizing, while the plot on the left shows the size of the buffers. The sudden hike in performance is attributable to increased cache hit-rate. Resizable buffers thus automatically adapt to cache sizes. Not visible in this figure is the computational overhead of the adaptive buffers. When compiled without optimisations we noticed up to a 10% slowdown compared to the static ring for large buffers.

4.2.2 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 [6].

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 with a producer and consumer reading packets at maximum rates. 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 fast 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 advantages of d-DBufs (seeking) are not evaluated in this test.

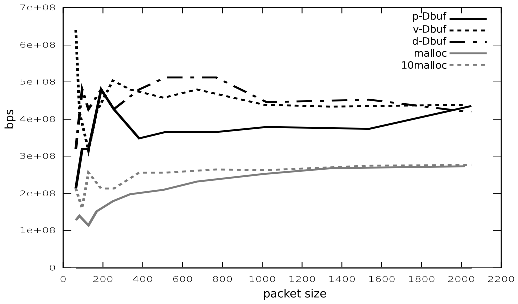


Figure 5: performance of different ring implementations

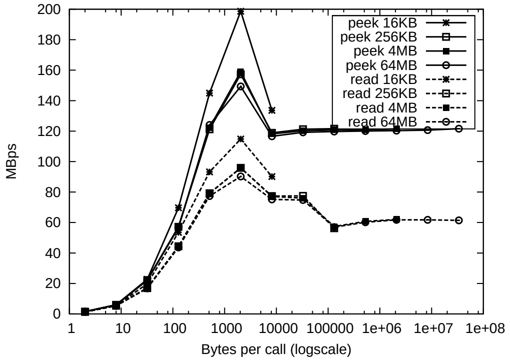


Figure 6: Impact of ring size and datablock size

Figure 6 shows the impact of buffer size and per-call block size on throughput. We show only the c-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. Obviously, we can not read more from a buffer than its size in bytes, hence the cut-offs towards the right. Again, the influence of cache sizes on throughput is clear.

An orthogonal method to increase cache utilisation is to partition blocks into multiple rings based on their size: buddy allocation. In buddy allocation each ring suffers from external waste independently, increasing overall external waste. Increased cache-utilisation for smaller blocks must offset this cost. Because the minimum block size in stream processing is in the order of 10s of bytes, however, few cachelines contain multiple blocks. Therefore the benefit does not outweighs the cost. By default, data buffers are implemented as d-DBufs, but MON applications use c-DBufs for TCP streams. FS uses c-DBufs for data and d-DBufs for packet headers.

4.3 Handling Resource Exhaustion

Because ringbuffers are statically allocated, they may suffer from memory exhaustion long before the VMM allocator fails. For this reason we must deal with out-of-memory (OOM) errors ourselves. We can employ two strategies: exhaustion avoidance and exhaustion resolution. Avoidance implies slowing the speed of the producer. This method needs a feedback circuit back to producers or a buffer-aware scheduler. Scout and Random Early Detection [14] employ such pressure reducing techniques. The advantage is that fewer cycles are spent processing blocks that will be dropped anyway, and thus more blocks can be processed completely.

4.3.1 Exhaustion Avoidance

Early dropping is not easily implemented, because *Beltway* has no central scheduler and bottlenecks can occur in buffers far from the input (e.g., an process-private IBuf). A contrived example would be to use a single-slot IBuf to index an n-slot DBuf. Unless the IBuf consumer is n times faster than the DBuf producer many blocks will go unused.

To minimise cycles wasted on intermediate steps we have evaluated an alternative to tail-drop that *gradually* pushes congestion to the source. Conceptually the algorithm works as follows: buffers artificially expand and contract the reported amount of available space based on fill-ratio and -rate. Buffers report at most the real amount of free space, but start underreporting when their buffer fills up. This early warning trickles back to the input, where packets are dropped that would otherwise have been dropped somewhere in the buffer network (but would not have reached their destination). By contracting gradually and starting early, bottlenecks are recursively passed on as the intermediate queues fill up. In the end bottlenecks are found at the inputs of the buffer network, and only in as far as these contribute to the network overload.

The problem of exhaustion avoidance is closely related to congestion avoidance in network queues, but is unique in that multiple *Beltway buffers* compete for the same exhaustible resources (CPU cycles). Our solution resembles Random Early Detection [14] with drop policy. The aims, environment and feedback mechanism differ from that of routers and RED, however. Packets are not dropped en-route; they are kept in-flight for longer.

Self-stabilisation depends strongly on the agility of stream expansion and contraction. This parameter is hard to configure. A random setting will probably remain highly unstable or converge to a suboptimal stable state. For this reason the default in *Beltway buffers* is still to use tail-drop. We mean to investigate stabilisation in the near future and then move to this model as default. Because the system is highly modular this will be an easy change to carry out.

4.3.2 Exhaustion Resolution

In practise, we can do without feedback, because we have also implemented *a posteriori* exhaustion resolution. In a ringbuffer an OOM condition occurs if the consumer is a whole buffer length behind the producer. One of the simplest resolution strategies, is to *block* producers until there is space in the buffer. However, we prefer not to block. Each space has a single processing loop, which benefits performance. Blocking would complicate this design to the detriment of throughput. A blocking *Beltway buffer* implementation exists, but is not used in practise.

Instead, we employ three resolution strategies that we call Slow Reader Preference (SRP), Medium Reader Preference (MRP), and Fast Reader Preference (FRP). The methods implement different choice points on the trade-off between high speeds and low droprate. 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 new arrivals. 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. In practise this is often not a problem as the number of consumers is typically small.

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. Resolution is thus simple: 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 that we noticed 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 incurs complexity of its own. It may lead to unused writes to the ring (which can be partly solved by exhaustion avoidance) and incurs per-block integrity checks by all consumers. On a miss we need to decide how many blocks are skipped. Moving the consumer's position forward by a single slot minimises the number of dropped blocks in the short term. But OOM handling is more expensive than normal processing, so it is often beneficial to move forward the position with multiple slots — perhaps even the entire ring — than to skip a single slot. This is another variable that is difficult to set at compile time.

One heuristic we employ to limit droprate is to increase stepsize quadratically on a miss, and to decrease linearly on a hit – similar to the Additive Increase, Multiplicative Decrease congestion control of TCP. Hits are considered common case and should be as fast as possible. For this reason we slightly alter the scheme to calculate only on a miss the distance to the last miss and from this recalculate the next stepsize.

Because an FRP-enabled buffer is completely lock-free, data can be overwritten while a consumer is accessing it. 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. FRP-aware applications will see higher throughput, because they can replace the copy cost in the general case with `peek(..)` plus less expensive bound 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) is a compromise 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. In practise, MRP amounts to SRP except for very tardy readers. The price we pay is potentially reduced efficiency as the producer is sometimes held up unnecessarily. On the other hand, in MRP slow readers cannot hold up fast readers, synchronisation is minimal, and behaviour is mostly similar to tail-drop.

By default *Beltway buffers* employ the SRP policy as it is more familiar to users, but administrators may configure the system as FRP or MRP. As disk I/O tends to be bottleneck, FS does not normally suffer from OOM. MON benefits significantly from MRP and even SRP for link rates below the processing capacity of the host.

4.4 Non-uniform Data Access

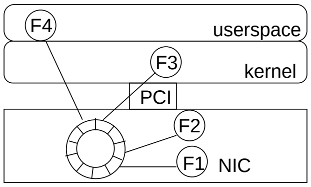


Figure 7: Remote spaces

Data in a buffer may be accessed by local consumers as well as by those reading over a communication channel. An example is shown in Figure 7, where the data in the NIC is processed by functions in kernel, userspace, and locally. Access characteristics are unequal across boundaries: a consumer operation local to the buffer has lower access costs than one that must cross the PCI bus, a situation known as non-uniform memory access (NUMA).

NUMA prompted *Beltway buffers* to specialise buffer access so that reads or writes across space boundaries can be handled differently from local reads and writes. *Beltway buffers*' presence in multiple spaces also requires buffer specialisations that overcome architectural differences, particularly regarding endianness. For instance, the *Beltway buffers* on the Intel IXP network processor cards are big-endian. For such buffers, metadata that is transferred by the card towards the little-endian host needs conversion prior to access. A buffer implementation converts the metadata *transparently to both the producer and consumer*.

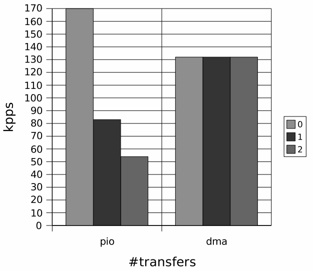


Figure 8: zero-copy vs dma

The optimal strategy for cross-space data sharing depends on the characteristics of the underlying hardware crossing and the access pattern of the client. **Zero-copy** access, whereby data is touched directly through remapping intuitively appears the optimal solution for general case processing. Zero-copy makes remote data randomly accessible. However, the mechanism fails to make use of hardware support, such as DMA or prefetching. In addition, reads across high-latency crossings incur roundtrip delays. Figure 8 plots *Beltway* performance of programmed I/O based zero-copy versus DMA-based copy-once for network packets received by an IXP2400 network processor card, with 0,1, or 2 accesses per packet. Whenever any data is touched, DMA wins. We advocate restricting zero-copy to space crossings for which no hardware support exist, spaces that access the same memory anyway (e.g., userspace and kernel), and to applications that access data sparsely and non-sequentially.

For high-speed sequential access – the common case in stream processing – we observed that copying the buffer using hardware support is cheaper: **copy once**. A drawback of this scheme is that it copies all data, also that which is not accessed at all. Overhead is commonly minimised (e.g., by pcap) by manually limiting copying to a subset of all data, such as IP headers. We have implemented an alternative strategy that uses caching.

In **cached copy** a replica allocates enough space to copy the entire original buffer, but fetches blocks only when requested. Note that cache blocksize does not have to be related to stream block size. In general it is smaller. Size and number of blocks transferred at once is highly dependent of hardware characteristics (e.g., support for burst-traffic).

The last in our set of interfaces is **copy on demand**, which combines zero copy with copy once. In stream processing it is common to have to inspect a small number of bytes (a header) to decide whether the rest of the data is of interest. For this situation it is preferable to use zero-copy to access the first bytes and to only incur the overhead of a full copy when explicitly requested to do so. The large copy can then use hardware support, without saturating a bottleneck link (e.g., the PCI bus) with unused data.

Traditionally, each peripheral driver must re-implement one of these strategies. *Beltway buffers* centralises the code, but not strategy selection. For each boundary between spaces, the most optimal implementation of data transfer is chosen. By default, *Beltway buffers* use copy-once across the PCI bus, cached-copy for network links and zero-copy across the kernel ABI. However, administrators or driver developers may override the default configurations and specify their own (e.g., to exploit application-specific knowledge, or for experimentation).

A practical issue concerns DMA efficiency and metadata updates. The PCI bus is a clocked bitpipe shared via bus arbitration. For efficiency, the bus should be used in burst mode, transferring the maximum number of bytes in one go. To accomplish this, *Beltway buffers* tries to transfer data in one direction only and to avoid bus arbitration. For instance, when network data is transferred from the network card to host memory, we prefer to handle metadata updates (e.g., incrementing read and write pointers) on the host side also. The network card will DMA the write pointer in its buffer to the host-side where a copy is maintained. The alternative, transferring read pointers from the host to the network card is less efficient and, if initiated by the host, requires frequent bus arbitration. Even a trickle of control information flowing towards the card may severely hurt performance of the card's DMA. Because of this, FRP is uniquely suited for crossing the PCI bridge.

5. PRACTICAL USE IN NETWORKING

To demonstrate practical use, we engineered the most popular traditional network interfaces to use *Beltway buffers*: the BSD Sockets API and the pcap library for packet filtering. In addition, we developed our own dataplane for generic communication paths, known as Streamline. Besides discussing these three systems, we show

how we used Streamline to implement in-place TCP reassembly.

5.1 PCAP

The pcap library implements an application programming interface for network packet capture. It is used by well-known MON tools like `tcpdump`, `nmap`, `Ethereal`, and `Snort`. As it provides little functionality beyond capturing packets, it is easily modified to use *Beltway buffers*. In *Beltway*, it is implemented on top of the POSIX API. We only summarise the most important calls.

Software built on the pcap library consists of three parts. First, we select a network interface (`pcap_open_live()`). Second, we apply a filter to select those packets in which we are interested (`pcap_compile()` and `pcap_setfilter()`). Third, pcap enters its main execution loop. We provide the `pcap_open` functionality by a small amount of code either in or on top of the network driver that is able to push incoming packets (up to the pcap-defined snap length) into a *Beltway* DBuf. Next, the Berkeley Packet Filter is inserted. This is the same code used by other pcap implementations, but we wrapped the call in code that reads from a DBuf and pushes selected packets to a separate IBuf. Pcap reads from this IBuf and adds a header in the pcap format, containing timestamps and other metadata.

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 from userspace and successively calling the callback function until there are no more packets available. In practise, monitoring of running applications is practically 'free'.

5.2 BSD Sockets

Compared to pcap, the socket API involves far more processing. Sockets abstract away tasks like (de-)multiplexing, (de-)fragmentation, reassembly, checksumming and header (un-)packing. In other words, rather than vanilla access to the *Beltway buffers*, the sockets API provides a full protocol stack. To support legacy applications we implemented the complete socket interface. The single change underlying all our socket calls is that they are executed in the process's context and thus incur no context switch. To accommodate this, processes have been given limited access to kernel state. Doing so is trivial and safe as long as it concerns metadata only and access is restricted to read-only for shared or otherwise guarded resources. Runtime performance benefits mostly from context-switch reductions in the datapath (through `recv`, `accept`, `select`/`poll` and `send`). For this reason we will not discuss the other calls in the interface (although they too are context switch-free).

The `accept` call is handled in userspace in the same manner as `read` and `write`: it waits on an IBuf that contains elements pointing to new flows. `recv`, `send` and related calls are even more similar: these are protocol-specific wrappers around `read` and `write` that prepend data with headers. All these calls are handled asynchronously, i.e., without the need to switch to kernel space for each call.

To wait on multiple file-descriptors, `select` and `poll` are used. Standard implementations of these have been shown to scale inefficiently [4]. To circumvent polling on all IBufs, we supply a separate IBuf into which all data destined for a process 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.

The current implementation is limited in the options supported. For instance, while non-blocking sockets are supported, we have not yet implemented other refinements. On the other hand, *Beltway*'s sockets implementation is more efficient than system call-based versions, because buffers are transparently shared.

5.3 Streamline

On top of *Beltway buffers* we also implemented a composable network stack known as Streamline. Streamline allows users to specify applications by clicking together processing steps (known as 'functions') in a directed acyclic graph (DAG)[2]. For instance, we can implement an x-kernel-like network subsystem with steps for handling and (de-)multiplexing different protocol data units [16]. Or we can click together the various steps involved in video decoding and rendering as proposed in [22]. Indeed, packet reception and preparation for the BSD Sockets interface is handled through a Streamline request, as is the loading of BPF for PCAP.

When the user instantiates the DAG all functions in the graph are automatically setup in the most appropriate space. Some functions may be instantiated on dedicated hardware, some in the kernel and some in userspace. Streamline handles the traversal of data through the *Beltway buffer* system. In TCP-based networking, this means that packets are pushed into a ring by a Streamline function at the bottom of the processing hierarchy, and subsequently pushed to an IP function for defragmentation, and on to a TCP reassembly function, etc.

To maximise cache-efficiency we must either process as much data from the network as possible with the same function or the other way around. Which method should be preferred depends on data access patterns in functions. We chose to exploit data-locality at the cost of instruction locality, because we expect that with a level-1 cache in the order of 16kB the majority of instructions will fit in the instruction cache. The data cache, on the other hand, is exhausted with even a handful of packets. We strive to process a block without causing duplicate cache misses – at least within the same space. In essence we try to apply the performance benefits of integrated layer processing [8] without throwing out the clean stream abstraction. The alternative would be to queue packets (or pointers) at each function. Queueing is computationally more costly than some functions themselves and reduces cache efficiency. Streamline's datapath makes direct use of *Beltway buffers*' support for optimising away IBufs within the same space.

A minimal data-aware processing loop must, for each block, pop a function from a call stack, call it, evaluate the result and selectively push dependent functions onto the stack. In corner cases the code will have to store the block in a DBuf or a reference in an IBuf. IBuf writes are limited to streams that are scheduled for direct inspection by the application and those that need to be mapped to other spaces. DBuf writes are only carried out when data is modified (and cannot be modified in-place), or when security policies obstruct the cross-space sharing of a buffer.

Asynchronous signalling is used extensively by stream-based architectures to forward traffic (e.g., by SEDA and Scout). However,

[2] Streamline is available for download from `ffpf.sf.net.`

signalling is far more costly than a hardcoded functional dataplane. Clark therefore proposed a structured architecture that replaces signalling with function calling [7]. We take the same approach, but extend it by reducing (amortised) function calling cost and tuning the forwarding to the environment. Concretely, we identify three forwarding methods, each at least an order faster than the next: (i) inlined functions, (ii) indirect functions, and (iii) signalling.

An inlined function call is cheapest, but the advantage of inlining is offset by increased codesize, which also increases instruction cache miss rate. Data-access costs can also increase, because registers are more scarce [1]. For this reason only a handful of trivial Streamline functions is inlined. Note that complete functional paths are suboptimal because they employ deep nesting. When evaluating call nesting overhead, we found that deep nesting significantly degrades performance because of the tens of cycles consumed by each call. For simple functions, the overhead of calling may even dominate the computation.

We use a switch to decide which method to use. Each inlined function has its own case statement in the switch, whereas others have a fall-through label that leads to the second case. Inlining thus incurs one extra lookup to un-inlined functions. The technique must therefore only be applied to oft-used functions. We use it for a TCP port-filter, a packet counter and similar routines that consist of at most 10 lines of C code.

Functions that are not selected for inlining, but exist in the same space as their predecessor, incur an indirect function call. Indirect calls are between one and two orders of magnitude more expensive than inlined code. The right plot of Figure 9 compares both calling methods with and without compiler optimisations turned on (10^9 iteration on a 1.8GHz P4). We limit this cost by keeping parameter count low. Even for moderate numbers of arguments up to half of the calling cost is due to stack manipulation. For this reason we rolled parameters that are only used by a subset of all functions into a structure. Individual lookups incur an additional memory lookup, but overall costs are lower. The remaining parameters (three) are passed through registers, removing all non-functional memory lookups from the dataplane.

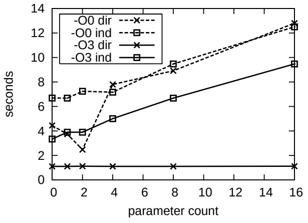


Figure 9: Call indirection and parameter count

Across spaces, direct function calls cannot be used. Here we must use a form of signalling. We try to cross boundaries as efficiently as possible (see also Section 4.4). For instance, a smart NIC may DMA data from the card to the host, while between the kernel and processes remapping is employed. Switching costs are reduced through signal coalescing. In principle transfer strategies are selected at compile-time, but defaults may be overridden. Streamline needs not worry about boundary traversal, as *Beltway buffers* take care of these details.

Due to careful examination and elimination of the root causes of overhead, processing is efficient. While the design is structured and modular, we do not sacrifice performance. In practise, Streamline is used for all network processing; together with *Beltway buffers* it implements a complete network stack. MON applications use Streamline for all sorts of functional elements (pattern matching, reassembly, etc.), while FS makes use mostly of the elements implementing the traditional network stack.

5.4 In-place TCP Stream Reassembly

Recreating a continuous stream of data from packets is expensive because in the common case it incurs a copy of the full payload. TCP is especially difficult to reconstruct, as it allows data to overlap and has many variants. Besides a traditional TCP implementation, Streamline offers a TCP reassembly module – that is heavily dependent on *Beltway buffers* – where streams are reassembled in-place, i.e. in *zero-copy* fashion. In terms of performance, we win by reducing memory-access costs. 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 a cost for bookkeeping of the start and length of each TCP segment. Due to the (growing) gap between memory and CPU speed this cost is substantially smaller.

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. With `peek()`, instead of allocating a transfer buffer and copying data into it we return a pointer directly into the original segment. As the standards do not prescribe one way or the other, in case of overlapping segments, we supply the bytes either from the first segment or from the second according to configuration. The system may also be configured to drop the overlapping segment (thus implementing a simple protocol scrubber).

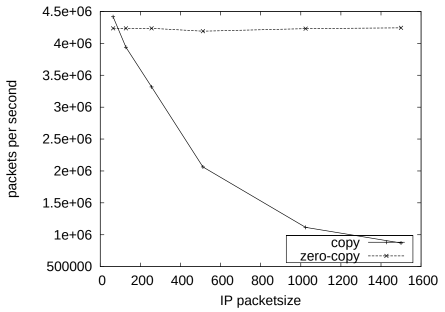
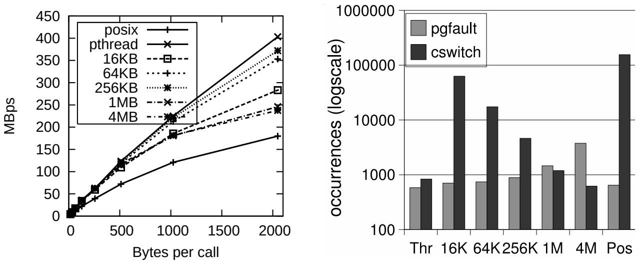


Figure 10: TCP reassembly

Independent of maximal obtainable throughput is the question how indirect stream reassembly measures up to regular copy-based reassembly. For this reason we have compared them head-to-head. The two functions share the majority of code, only differing in their actual data bookkeeping methods. Figure 10 shows that indirect reassembly outperforms copy-based reassembly in the general case. Only for the smallest packets can the computational overhead of extra indirection be seen.

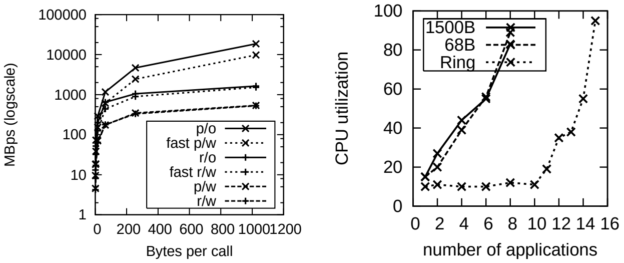
6. EVALUATION

In previous sections, we presented micro-benchmarks to support specific design decisions. We now compare Streamline's interfaces head-to-head to existing solutions. All tests were executed on an AMD Sempron 3000+ with 128KB of (unified) L2 cache memory running Linux 2.6.16 and Streamline 1.7.0.



(a) pipe performance (b) pipe overhead

Figure 11: Evaluation of pipes



(a) Read and write performance (b) Load with multiple consumers

Figure 12: Reading, writing and multiple consumers

We compare *Beltway buffers* head-to-head to Linux in terms of copy costs. Figure 11(a) shows comparative performance of straightforward copying through a UNIX pipe. We do not use the peek optimisation and thus copy the same amount of data. Any performance improvement comes from a reduction in context switching. As upper bound on the achievable performance we also show results of a threaded application with direct access to shared memory. This application outperforms Linux pipes by a factor of 2.5. In between are 5 differently sized SRP c-DBufs. We see that the fastest implementation has neither the largest, nor the smallest ring. Initially, performance grows with the buffer. This is to be expected as the number of context switches drops when calls are less likely to block. Ultimately, however, page-faults affect performance when the TLB runs out of entries. Peak throughput occurs with rings sized between 64 and 256KB. Within this range the number of context switches diminishes gradually, while we do not yet witness TLB thrashing. The same behaviour also holds for the threaded application, but we only show its optimal result for clarity.

Orthogonal to ring size is the call size: the number of bytes per function call. For calls with small blocks CPU processing is the main overhead. Here context switch reduction is most beneficial. In Figure 11(b) we plot the number of context switches and pagefaults independently (aggregated over all block sizes). As expected, context switch count diminishes exponentially. From 256KB onwards page-fault occurrences grow rapidly.

The pipe test showed that *Beltway buffers* are more efficient in copying than traditional Linux systems. We now take a closer look at some optimisations: we compare performance of peek to that of read, with and without the fast splice optimisation. We leave the default Linux implementation of read/write out of the test, because we just saw that it is slower than our read/write implementation.

Figure 12(a) shows the performance of reading and writing data for an FS application, where data is read from the page cache (which is filled automatically by means of prefetching) and written to a network transmit IBuf. The fastest mechanism is p/o: the peek equivalent of read-only access. This mechanism processes even faster than the physical bus permits, because it doesn't actually touch any data. Note that read-only access is not very useful and only shown as upper boundary on achievable performance.

Barely slower is fast peek/write, which combines peek with the fast splice optimisation. This, too, does not actually touch any data, but it must write out an IBuf element. Overhead caused by read can be seen by comparing these two results with the next 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. Then throughput drops again, to a third. We can tell 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.

We have argued that especially in MON practice, there is a growing need for multiple applications accessing the same data (end applications, IDS, profilers, etc.). While current systems incur copies for each application, *Beltway buffers* are capable of sharing buffers horizontally also. Our next experiment evaluates this aspect.

Figure 12(b) shows performance of tcpdump, one of the most popular monitoring applications in use. We compare standard tcpdump, which uses Linux Socket Filters, to a version that talks to our PCAP interface. We compare throughput for a moderate datastream: 100 MBit of large sized packets per second (i.e, 75000 pps). This allows us to investigate scalability. 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. Most processing is coalesced in the kernel by Streamline. Furthermore, we have configured tcpdump to output minimal information. Still, the application must execute code to generate log messages. This visualisation cost is apparently insignificant.

With more than 10 applications thrashing occurs. By inspecting the number of voluntary and forced context switches independently we learnt that, although involuntary switching increases for each extra application, thrashing does not occur until the number of *voluntary* switches starts decreasing. That is a sign that applications do not get the chance to handle all data when it comes in, but need an extra time-slot. This is the beginning of a snowball effect: thrashing.

Although we expected standard tcpdump to be memory bound, Figure 12(b) 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 will exist when accessing non-overlapping data, the common case in other than MON applications.

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 TCP reassembly, regular expression scanning of the entire payload, as well as a new type of protocol-specific malware checks. Despite the computational overhead, the system achieved a throughput of almost 1 Gbps in the worst case. Interested readers are referred to [10].

7. CONCLUSIONS

We have described *Beltway buffers*, a structured buffer subsystem geared for speed and flexibility. *Beltway buffers* place data in shared ring buffers that are indexed by universal pointers in smaller rings. The system offers IO paths with attractive caching properties and a datapath with limited copying and virtually no allocation overhead. In addition, it provides a solution for integrating the address spaces of multiple devices and execution spaces in a uniform buffer system. At the same time, the *Beltway buffer* system provides a familiar POSIX file API to its users, which makes it a natural fit for Unix-like operating systems.

Acknowledgements

We would like to thank Peter Druschel, Philip Homburg and Kees Verstoep for commenting on earlier versions of this paper. We are grateful to Tomas Hruby for implementing *Beltway buffers* on Intel IXP network processors,

8. REFERENCES

[1] B. Ahlgren, M. Bjoerkman, and P. Gunningberg. The applicability of integrated layer processing. *IEEE JSAC*, 16(3):317–331, Apr. 1998.

[2] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proc. OSDI'04*, pages 347–363, San Francisco, December 2004.

[3] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Operating Systems Design and Implementation*, pages 277–291, 1996.

[4] A. Chandra and D. Mosberger. Scalability of linux Event-Dispatch mechanisms. In *USENIX Annual Technical Conference 2001*, pages 231–244, 2001.

[5] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, pages 333–346, Boston, Massachusetts, June 2001.

[6] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an internet backbone. In *INET'98*, Geneva, Switzerland, July 1998.

[7] D. D. Clark. The structuring of systems using upcalls. In *Proc. of SOSP'85*, pages 171–180, 1985.

[8] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[9] 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.

[10] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. Safecard: a gigabit IPS on the network card. In *Proc. of RAID'06*.

[11] L. Deri. Improving passive packet capture:beyond device polling. In *4th International System Administration and Network Engineering Conference (SANE'04)*, Amsterdam, NL, September 2004.

[12] P. Druschel, M. Abbott, M. Pagals, and L. Peterson. Network subsystems design. *IEEE Network*, 7(4):8–17, 1993.

[13] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[15] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. of SOSP'91*, pages 68–80, 1991.

[16] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[17] V. Jacobson and B. Felderman. Speeding up networking. Presented at Linux.conf.au, `http://iou.parisc-linux.org/lca2006/TCP_LCA2006.odp`, January 2006.

[18] S. Karlin and L. Peterson. VERA: an extensible router architecture. *Computer Networks*, 38(3):277–293, 2002.

[19] L. McVoy. The splice I/O model. `www.bitmover.com/lm/papers/splice.ps`, 1998.

[20] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[21] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Operating Systems Design and Implementation*, pages 153–167, 1996.

[22] M. Nijhuis, H. Bos, and H. Bal. Supporting reconfigurable parallel multimedia applications. In *Euro-Par'06 (distinguished paper)*, Dresden, Germany, August 2006.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[24] J. Paisley and J. Sventek. Real-time detection of grid bulk transfer traffic. In *Proc. of IEEE/IFIP NOMS'06*, Vancouver, Canada, April 2006.

[25] J. Pasquale and E. Anderson. Container shipping: Operating system support for I/O-intensive applications. *Computer*, 27(3):84–93, 1994.

[26] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[27] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of SOSP*, pages 230–243, 2001.