

# Streamline: Efficient OS Communication Through Versatile Streams

Willem de Bruijn and Herbert Bos  
 Vrije Universiteit Amsterdam  
 {wdb,herbertb}@few.vu.nl

## Abstract

Streamline is a stream-based communication subsystem that spans from embedded hardware to userspace processes. It improves performance of memory and I/O bound applications by constructing tailor-made datapaths for each application at runtime. Datapath optimisation removes unnecessary copying, context switching and cache replacement. It simplifies integration of embedded and distributed hardware. Streamline automates datapath optimisation and only presents users a clear, concise request language. Streamline exports the BSD sockets, POSIX file & pipe and PCAP interfaces. Observed performance improvements compared to Linux are up to an order of magnitude.

## 1 Introduction

Operating system overhead limits throughput of I/O bound applications, such as web- and mediaservers. Our goal is to increase performance for this important class of applications by (1) removing bottlenecks and (2) incorporating all available resources, specifically embedded and distributed hardware. We propose constructing datapaths on-demand from reusable modules to achieve both.

Current operating systems, instead of exhibiting a clean structure, comprise multiple weakly connected subsystems. Performance problems arise from excessive copying between drivers, OS kernel and applications; from wasting cycles and evicting useful cachelines due to interrupts and context switching; and from inadequate integration of embedded hardware, which is often limited to supporting niche applications through non-standard APIs. Over the years, the interfaces have become further muddled with ad-hoc optimizations. Such patchworks harm performance. For instance, a simple task such as reading a file from disk and writing it unmodified to the network (a common procedure in servers) incurs unnecessary context switching and copying to a process that immediately copies the data back to the kernel for transmission. In the process, we gum up the TLB, adversely affecting performance even further. Ad-hoc fixes, such as adding a `sendfile()` system call, alleviate this problem only in limited scenarios; many more servers transmit data

blocks than whole files. As such, it is a bit of a hack.

To reduce overhead and incorporate specialized hardware for all applications we developed Streamline: an extensible communication subsystem that outperforms Linux I/O by up to an order of magnitude. Streamline is designed to handle all I/O and provides the common files, pipes and sockets abstractions. In addition, it supports many advanced features (e.g., user-accessible disk caches, multiway named pipes, copy avoidance, prefetching, interrupt mitigation, and splicing [27]). Internally, the software is structured quite differently from existing systems, in a way that is attractive both for performance and for OS structuring.

To the best of our knowledge, we are the first to build a complete communication subsystem based solely on statically mapped ringbuffers that each accommodate multiple blocks (rather than buffering and mapping per data block). While ring-based communication is not new (see Govindan’s work on continuous media [19]), most existing projects have limited themselves to partial solutions centered around a single ring. They offer no general solution for issues like protection, memory exhaustion, etc. As we will see, a single ring is not sufficient for many reasons. For instance, as much as possible a buffer should be tied to a processor for caching reasons (processor affinity), which suggests to have at least as many buffers as processors serving them. In addition, many network devices already use their own buffers, like it or not. To our knowledge no existing system has succeeded in building an integrated communication subsystem out of many statically allocated rings.

The contribution of this paper then is that we are the first to explore how to build a mature (stream-based) communication subsystem on top of multiple ring buffers, preserving the efficiency of rings without sacrificing usability or extensibility. By making buffers resizable, we strive to eliminate the problem of inefficient memory usage typically associated with ring buffers. Furthermore, through automated path construction, Streamline is able to shield the user from most technical details. We incorporate all hardware seamlessly into the datapath through the concept of ‘execution spaces’ that reside in processes, the OS kernel, embedded devices and on remote hosts. Furthermore, we maximise I/O throughput by (a) pushing operations to the most suitable execution space, (b) sharing intermediate results, and

through (c) copy-, context switch-, cache eviction- and allocation avoidance.

On top of the buffering plane, Streamline layers a stream-oriented communication subsystem. We replace the static OS datapath with lightweight paths optimised for each individual application. Streamline assembles paths on demand from the most efficient processing, forwarding and copying methods at hand. Construction from exchangeable parts enables software experimentation and hardware adaptation, both of which benefit performance. Structured OS design has previously been shown to reduce complexity and increase performance [10, 21, 28]. Compared to earlier work in this area (which includes STREAMS [31], x-kernel [21], Scout [28], Click [24], FBufs [16], and I/O-Lite [29]), Streamline reduces runtime copy and allocation overhead through the ring-based buffering subsystem, a reduction in context switching through (adaptive) interrupt mitigation, and integration of embedded hardware through path optimization.

Aspects of Streamline build on the XXX filter framework [34], which revolved around a single shared packet buffer. After extending it to run on embedded hardware and span distributed machines, we redesigned the system from scratch to deal continuous streams and applications beyond filtering. To do so, we had to solve the problem of building a full networking subsystem on multiple shared ring buffers, while preserving efficiency, isolation, and usability. It now supports also non-networking functionality like Unix pipes, disk access, page caching, as well as novel concepts like *splice* [27] and multiway pipes.

While this is the first time we submit a full description of Streamline for publication, the system previously served as a basis for a set of challenging applications, among which a token based switch for access to optical networks and a gigabit intrusion prevention system on a network card [35]. The reference implementation runs on a large set of machine architectures and scales from single embedded boards up to clusters. Commercially, it is used in a third party network monitoring card. As a stable system that has been shown to carry real-world applications efficiently, Streamline strengthens the arguments in favour of OS structuring.

The remainder of this paper is organised as follows. In Section 2, we compare Streamline to related work. The main concepts are introduced in Section 3. Construction of the lightweight paths is the topic of Section 4. the functional datapath is described in Section 5, and the buffer management system in Section 6. Section 7 details implementational details of interfaces and hardware support. We evaluate the system in Section 8 and draw conclusions in Section 9.

## 2 Related Work

Structured OS design is now new. STREAMS [31] interconnects functional modules through full-duplex messages queues (“streams”) in a vertical stack. Full-duplex process-

ing forces each module to have an upward and downward task: a one-to-one mapping not always present in protocols. Layer crosstalk and sessions complicate compartmentalisation. A stack oversimplifies the control flow in networking code. Later versions added limited support for multiplexing. STREAMS is less efficient than a handcrafted datapath, mainly because of its inter-module transfer overhead.

The dataflow model of computation (MoC) remained popular for OS structuring. The x-Kernel [21] adds session handling. Processing extends across protection domains, a feature added in anticipation of microkernel uptake. Overhead is minimised by using procedure calling between modules – the performance benefits of which had just been shown [10]. The x-Kernel was extended into Scout [28], which ensures QoS constraints through explicit module scheduling and extends the stack into domain-specific processing. Scout has since been ported to the Linux kernel [4]. More recent is SEDA [33], a dataflow-based server, which uses an explicit store & forward network within the server. The Dataflow MoC was further applied to routing with the Click router [24]. Click paths are static; contrary to the xKernel, Scout, SEDA and Streamline, Click can remain unaware of sessions and transport-layer streams, because it deals with layers 3 and below. A recent addition, Dryad [22], maps an operation graph onto a cluster. Like Streamline, it automatically maps a request to resources, but the systems tackle orthogonal issues. Streamline focuses on OS bottlenecks and embedded devices, Dryad on distributed operation.

Streamline extends the stream-based processing model in three ways: it reduces dataflow related overhead, incorporates embedded hardware and automates both, exporting to the user only a concise, high-level request language.

**Packet Filtering** Streamline replaces protocol-based path selection with user-controlled selection and modification operators. In this sense it is related to packet filters (from which it originated). Streamline incorporates existing languages, such as BPF [26], as functional modules and extends the state-of-the-art with FPL, a stateful packet filter that is compiled at runtime to various hardware architectures. Windmills [25] and BPF+ [5] are similar, but lack Streamline’s hardware selection and code-shipping interface. Like Pathfinder [3] and DPF [17], Streamline enforces prefix-sharing between overlapping requests.

**Copy and Context Switch Avoidance** The two major causes of non-functional overhead in communication paths are data copying and context switching. 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 a cost: recurrent modifications to VM structures reduce efficiency. Bruscolino [7] categorised previous efforts and showed them to perform roughly identical. Druschel *et al.* describe copy

avoidance ideas for network buffers [15] and subsequently translate these into Fbufs [16]. Fbufs are later incorporated in IO-Lite [29], which replaces copying and copy-on-write system-wide with mutable pointer structures to immutable buffers. Streamline eschews block-level allocation. We amortise overhead by moving all blocks into shared ring buffers [19], which enables interrupt mitigation across protection domains.

**Hardware Integration** OS structuring has mostly been limited to multitasking OSEs on uniprocessors. Incorporation of non-standard hardware, such as cryptographic units, programmable NICs, multicore CPUs and clusters is lacking. Scout was used together with a programmable NIC to build an extensible layered router, Vera [23]. Like Click, Vera only works on packets. Each execution layer’s implementation is complex and handcrafted. Extensibility does not reach the dataplane layer. Spin [6] moves application logic to the kernel. Spine [18] extends these concepts to programmable network cards. The solution is similar to how Streamline pushes computations to NICs, but Spine lacks the generality of dataflow processing.

**Stream Processing** The dataflow MoC is not limited to network processing. It also underlies stream databases (SDs) [1, 8, 13, 20, 32], which differ in application domain (e.g., financial transactions) and operate on a different timescale, using disk-storage.

### 3 Design

We define a **stream** as *all traffic that matches any programmable criteria*. Users specify the criteria as an operation over all input, which is expressive enough to support all known network tasks (say, TCP reassembly, decryption, etc.) and more. The operation decomposes into smaller modules, and we encode operation requests as composite directed acyclic graphs, or **DAGs**. Expressing I/O processing as a DAG of connected operations is intuitive and corresponds to the way we tend to think about protocols. Vertices depict active **functional elements** (FEs) and edges passive **streams**. We call FEs active because they act on passing data: classifying, modifying and forwarding it. Streamline FEs are not pure kernels, because they can access global state. For example, TCP reassembly needs to account flow-state. Streamline sets up stateful memory when it connects an FE to a vertex in the DAG, during **instantiation**. The relation between an FE and its **instance** resembles that of a class and its object, but FEs are simpler, lacking inheritance and polymorphism. Streams themselves are simply ordered blocks of data.

Requests are positioned over a model of the hard- and software environment to create a runnable task. The model

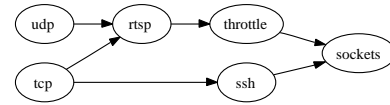


Figure 1: Partially nested requests

consists of a collection of execution **spaces** connected by communication **channels**.

A space is a self-contained environment in which FEs execute. This definition is broad enough to cover anything from cryptography assistance chips to userspace processes. Spaces are thus strongly heterogeneous: they can be programmable and have access to libraries, but this is no prerequisite. We achieve uniformity by defining a minimal interface that each space exports to FEs. Between many spaces (e.g., POSIX process, Linux kernel module) porting an FE is simply a matter of replacing boilerplate wrappers.

Channels forward all traffic between spaces. Streams are implemented over channels, as is the message-passing control plane. The combination of spaces and channels make up the **processing stack**

#### 3.1 Language

One of our main contributions towards ease-of-use is a concise language for expressing requests: the Streamline Request Language (SRL). SRL differs from traditional communication APIs by defining a textual representation for requests. We need a flexible, function-agnostic interface to carry out domain-specific operations for a wide range of application domains. SRL extends Unix pipes with task parallelism, classification and boolean selection. For networking applications the stream notation is a natural match. For example:

```
(tcp) > (http) > (file)
```

This request reassembles all tcp streams, selects only HTTP messages and writes these to separate files. Streamline uses the greater than symbol to represent a dependency between two processes, not a pipe – which describes task parallelism, for example for a firewall that drops all but a few protocols:

```
(tcp) > ( (http) | (ssh) | (rtsp) )
```

The example only accepts blocks that are accepted by (at least) one of the three filters. SRL also supports nesting: each node may be replaced by another SRL request; brackets can be used to remove ambiguity. SRL can pass optional parameters to an FE in the form of key=value pairs.

As presented, SRL cannot express partially nested graphs, of which Figure 1 shows an example. Such graphs, loops with additional input to an internal node (here `rtsp`), proved common. We resolve the issue by tagging nodes. By setting a ‘name’ option, a node can be referred to later on, giving:

```
(tcp) > ( (rtsp,name=tag) | (ssh) )
> (sockets) | (udp) > (tag)
```

**Classification** All examples so far used binary filtering: either data *is* tcp or it is not. Some tasks need more fine-grained selection. Port filtering would be tedious to specify if you have to define a stream for each individual port. Instead, we allow up to 16-bit selection granularity. For example, to restrict traffic to port 22, you write `(port) 22>`. Similarly, all other data is reflected using `(port) !80>` (`ids`); the exclamation mark inverts the condition. Ranges are also supported, as in `22:23>` and `!1:1023>`.

**Boolean Selection** Classification is supplemented with boolean selection. `(tcp)` and `(rtsp)` returns traffic that is both TCP and RTSP, the intersection of the two streams. `(tcp) or (rtsp)` returns traffic that is either one, the join.

Boolean operators demand processing. ‘Not’ substracts one stream from another, ‘and’ calculates the intersection and ‘or’ the union of two streams. The SRL parser rewrites the operations into FEs. A union is an FE that takes  $n$  inputs and only passes the first occurrence of a block. Intersection is similar, but instead of the first observed block, the  $n^{th}$  is passed, whereby  $n$  is the number of parallel streams. Boolean inversion is rewritten as condition inversion on the outgoing stream. Thus `not (tcp)` becomes `(tcp) !>`.

**Translation** Streamline also allows translation of an expression option. For example, BPF accepts a high-level expression, such as “ip port 80”, which the BPF interpreter cannot run directly. The expression must be translated into bytecode. For each FE  $f$  in a request, a translator is searched for that can translate the input pair  $(f, e)$  into  $(f', e')$ . Afterwards, the request may no longer be human readable as options may contain binary data – as will be the case for BPF.

### 3.2 Architecture

Now that the terminology is clear we move on to how these concepts materialise. Three concepts underlie the Streamline software architecture.

**Tailor-made Datapath** Streamline custom-builds each application’s datapath. Our assumption is that the burden of construction is more than made up for by the performance gain given by a more compact, efficient datapath. As a streamline path does not have to be general purpose, it gets by with fewer control-flow statements per packet. Statement-for-statement, constructed paths are slower than their compiled equivalents, however. We chose against code-generation, because that is infeasible when supporting third-party applications. Embedded hardware SDKs are also uncooperative. Moreover, the cost turns out to be small if

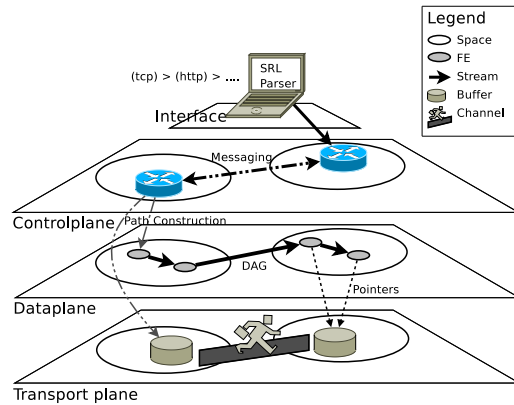


Figure 2: Architecture

streams are implemented correctly (we quantify this in Section 5). In the few exceptional cases where a constructed datapath causes noticeable slowdown, we use special-purpose state-machines, such as BPF, as FEs within the DAG.

**Specialised Implementations** When technical details are hidden behind a uniform interface, selection can be based on other priorities. What is more, it can be automated. An FPGA shares few characteristics with a Fortran function, but if both export the same interface for the same task (e.g., `(aes)`), a program such as Streamline can compare them purely on efficiency.

Aware of the performance afflictions attributed to stream-based interfaces and other OS abstractions (e.g., microkernels), we designed Streamline from the start with speed in mind. Generic interfaces reconciles high performance with ease of use, because it takes the advantage of hand-optimisation (efficiency) without the costs (complexity, time consumption, bug proneness).

**Separation of Concerns** FEs and streams are natural concepts for reasoning about datapaths. But the simplicity is largely an illusion. Underneath, pipes and buffers make up quite a different structure, and Streamline explicitly distinguished between the *processing plane* (containing the graph of connected FEs) and the underlying *buffering plane* (responsible for storing, mapping, and possibly copying data blocks in ring buffers). Both are administered from a distinct *control plane*. Control logic can be further split in two layers: the upper level SRL parser, which breaks compound requests into micro-tasks and lower level code that carries out resource discovery, selection and allocation.

The layered architecture is shown placed over a few spaces in Figure 2. When the SRL parser receives a request it breaks the DAG up into modules. Individual module requests are sent through the controlplane to all spaces. Each space has its own controller that constructs paths from the basic functional elements: FEs, streams, buffers and paths.

There are more processing than buffering elements, because FEs share references to the same data. We now continue by inspecting each plane individually.

## 4 The Controlplane

Generating an optimal path through an unknown set of resources and catching all runtime exceptions is error-prone. Rendering the construction transparent to the user is therefore essential. Control logic carries out the three steps involved in request adaptation: resource discovery, selection and allocation.

**Resource Discovery** For each FE in a request an implementation must be found in one of the spaces. For instance, a pattern recognition FE may be instantiated on the NIC in content addressable memory or in a userspace application. Lookup is distributed: each space keeps a local registry of available FE implementations. Requests are broadcast to all spaces from the caller's space through a messaging subsystem. Within a DAG, FE discovery is serialised. If no implementation can be found for one of the FEs the entire process is halted.

**Resource Selection** When a discovery request returns multiple candidate implementations a selection must be made. Datapath efficiency and throughput are dependent on the chosen FE implementation, cost of forwarding through the buffering plane and balance of load across spaces. The three given metrics are not orthogonal. One space may have a fast hardware accelerated FE implementation, but moving an FE there is counter-productive if doing so introduces excessive transport cost.

We eschew complex optimisation in favour of a simple heuristic: *push processing down*. Spaces are connected more-or-less vertically, from software-isolated applications at the top down to embedded devices such as FPGAs and NICs at the bottom. We assume that the further down this processing stack a calculation is pushed, the cheaper it becomes. Close to the network many DAGs will overlap and some traffic (e.g., malicious network data) can be shed early, reducing path traversal cost. Also, For each FE the "lowest" available implementation is thus selected.

**Resource Allocation** The resource selection algorithm returns micro-allocation requests that are sent to their destination through the messaging plane. As the allocation takes place across multiple spaces chance of partial failure is high. To keep the environment from entering an undefined state, allocations must be transactional. The key to achieving transactionality is to have small steps with only binary outcome, and to make each step reversible.

Streamline allocation is split into 3 phases. First, FEs are instantiated into runnable objects. If and only if all FEs have

been instantiated they are connected through paths. Again, each path allocation may fail, causing a full reversal of all previous steps. Finally, all instances are activated so that data can start flowing. Instances must be activated from the network's outputs to its inputs so that all are ready when the first data arrives.

**Implementation Details** Both discovery and allocation use message passing to forward requests. Because spaces may be very confined, they cannot be expected to implement a full IP stack. As alternative, we use the buffering plane to also forward control data. Control channels are bootstrapped when a space initiates. Application spaces are hard-coded to contact the host kernel, as are those of embedded devices. Further links, e.g., to remote hosts, must be connected manually.

DAGs often share overlapping graph fragments. Especially close to the data sources and within common network stacks can many instances be shared. Code sharing is implemented by merging requests from the sources onward. An instance is shared between flows if all its inputs are shared and the instance is stateless or in its initialisation state. When an FE request matches an active FE during instantiation, it silently attaches to this instance.

## 5 The Processing Plane

The processing plane executes FEs to manipulate streams. FEs manipulate streams by processing the constituent blocks in-order. They are translators, mapping input streams to output streams. As a stream can be of infinite length FEs can only access a subset at any time, which we refer to as a single *block*.

FEs manipulate streams in two ways: through block classification and -transformation. For each block an FE calculates a 16-bit classifier. Streamline compares this value with the ranges accepted by dependent FEs to decide whether to forward. Classification only prunes streams, transformation directly changes data. Minor transformations are performed in place (e.g., network address translation), but more complex modifications, such as TCP reassembly, transcend block boundaries. Here, no relationship between input and output flow can be presumed.

**Copy-free Stream Multiplexing** To support this decoupling, FEs must add and remove streams at runtime. We cannot ask users to specify each possible TCP stream separately in the DAG. More reasonable is to have them connect the TCP FE to the next stage in the processing once (e.g., *(tcp) > (http)*) and forward all streams produced by the TCP FE to dependent FEs of the original stream (which is usually dropped). To make this of any use, dependent FEs have to recognise individual streams. One solution is to

duplicate FEs along with streams, but that leads to a state-explosion. Instead, we use the classification value to communicate this information.

We call this mechanism *stream multiplexing*. Layer-3 routers, including those based on streams such as Click, have no need for it because they act only on discrete packets. Compiled network stacks limit multiplexing to a few hard-coded protocols, where copying is necessary to execute the transformation. This limits selection; you cannot, for example, easily save all HTTP streams. In Streamline the expression  $(tcp) > (http) > (disk)$  accomplishes just that.

Stream multiplexing has a second benefit when combined with IBufs. The classifier in an index groups blocks into streams. Offset and length fields allow selection of a sub-block. Together, they make it possible to discard outer-layer network headers without copying. This way we managed to implement full copy-free TCP stream reassembly. Out-of-order packet arrival and overlapping segments posed some hurdles, but through accounting of tcp-window state we also deal with these corner cases. Zero-copy TCP reassembly is described in more detail elsewhere [35].

**Intra-space FE Invocation: functions** FE invocation has a great influence on overall performance because it is carried out for every block and every FE. To process at high rates we strive to reduce this oft-executed overhead. Cache utilisation is maximised by either invoking the same FE for many blocks or by shepherding the same data through many FEs at once. We chose the latter, exploiting data-locality at the cost of instruction locality. With a level-1 cache in the order of 16kB the majority of instructions will fit in the instruction cache, while the data cache is exhausted with even a handful of network packets. We strive to process a block without causing duplicate cache-misses – at least within the same space. In essence we apply the performance benefits of Integrated Layer Processing (ILP) [11] without throwing out the stream abstraction.

A minimal data-oriented processing loop iteratively pops an FE from the call stack, executes it, evaluates the result and pushes dependent FEs onto the stack. In corner cases the program will also have to save (a reference to) the block, e.g., for inspection by the application. The only *practical* difference between a stream-based and a compiled datapath lies in the implementation of these steps. A compiled datapath uses the hardware stack, while a stream-based implementation implements a stack machine in software. A software stack is a necessity for us, because Streamline must be able to invoke FEs in remote spaces.

Asynchronous signalling is used extensively by stream-based architectures to forward traffic (e.g., by SEDA [33] and Scout [28]). Signalling

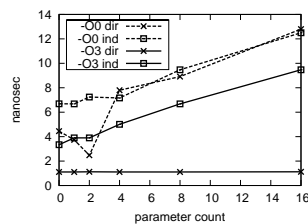


Figure 3: Call Overhead

is far costlier than function calling, however.

Clark proposed a structured architecture that replaces signalling with function calling [10]. Our calling scheme is a refined version of this. Because a general method is only so fast as the slowest underlying hardware, using the same calling method throughout a distributed system would be inefficient. Within Streamline we identify 3 dispatch methods, each at least an order faster than the next.

An inlined function call costs the least number of CPU cycles, but the advantage of inlining is offset by increased codesize and thus increased instruction cache miss-rate. Data-access costs may also increase, because registers are more scarce [2].

FEs that are not selected for inlining, but do execute in the same space as their predecessor in the DAG are reached through an indirect function call. Indirect calls are an order more expensive than inlined code. Figure 3 shows overhead per function call for four scenarios on a 1.8 GHz Pentium4. Compared to direct calls, indirect calling is between 4 and 12 times as slow, depending on compiler flags and number of arguments that have to be passed on the stack. Half of all function calling cost is due to stack manipulation. We limit this cost by keeping parameter count low. Concretely, we roll all parameters that are used only by a subset of all FEs into a structure and pass the remaining parameters through registers to remove all non-functional memory lookups from the processing plane.

We use a switch to decide which invocation 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 is therefore only 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.

**Inter-space FE invocation: adaptive signalling** Across spaces (hardware) function calls cannot be used. Instead, we resort to sending slower, costlier signals. High switching overhead can be stemmed with interrupt mitigation, whereby a signal is only sent for every  $n$ th packet or once per time interval. Recent Linux kernels use an adaptive interrupt mitigation scheme between the kernel and NICs called NAPI: the kernel disables interrupts as it processes the backlog of packets. Streamline moves between three levels of interrupting depending on datarate: signal-free, mitigate and signal-all. By default, Streamline uses a combined interval and datarate mitigation scheme, whereby a signal is sent only when either a timeout occurs or block-threshold is reached, but when the rate exceeds an upper threshold we disable signals altogether. Below a lower threshold we start signalling for each block. At low CPU loads we prefer to optimize latency over throughput.

## 6 The Buffering Plane

Until now we have portrayed streams simply as blocks flowing along the datapath. A literal implementation of this idea, a store-and-forward network, is inefficient, because it causes a copy at every function. Copy overhead can be reduced by queueing pointers instead of blocks themselves, as Scout [28] and STREAMS [31] do. Streamline does the same, but also culls unnecessary queues and prioritizes paths based on the substrate (e.g., local RAM, PCI, ethernet). This results in a transport graph quite distinct from the datapath. Analogously, we introduce a distinct layer of abstraction that handles data movement with no regards to higher layer processing: the *buffering plane*. This plane helps us integrate all hardware, avoid costly copying and setup allocation-free datapaths. We consider this one our main contributions.

**Ring Buffers** Most operating systems implement dynamic allocation: a memory region must be acquired for each block, either from the general allocator or a type-specific memory pool. In contrast, we statically allocate ring buffers capable of holding many blocks. Rings hold three advantages over dynamic allocation. *Offline allocation*: for light processing at high rates runtime allocation is unaffordable. Rings are allocated statically. *Scale-free sharing*: even if copying can be replaced with page sharing, per-block solutions must bear mapping cost with every block. In contrast, rings can be shared permanently. *Lock-free sharing*: rings remove the need for locking between concurrent consumers. We will see that *specialised* rings remove other costly types of locking (e.g., across a peripheral bus).

Govindan [19] used rings as one-to-one pipes between a process and the kernel. Our buffering plane extends this and provides sharing of arbitrary groups of rings between arbitrary spaces while hiding substrate peculiarities.

**Collections of Rings** Battling complexity, a single ring would be ideal. Unfortunately, that fails even for the simplest of tasks. First, unacceptable security concerns arise when there is no control over who has read and write access to the ring. Non-privileged processes, for example, may only inspect streams destined for their own applications.

Second, in the presence of multiple processors or cores, coupling a buffer to a single core helps performance, primarily because it keeps the data in a single cache. This is known as processor affinity. In contrast, a single ring is shared by all processors adversely affecting cache performance. 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.

Third, embedded resources complicate the memory architecture. They can use on-board buffers [35] or directly modify host memory [12]. An integrated buffering system must provide a single coherent view on the data and choose the

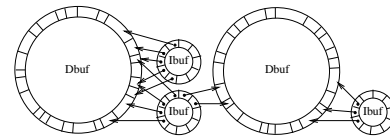


Figure 4: Chain of data and index buffers

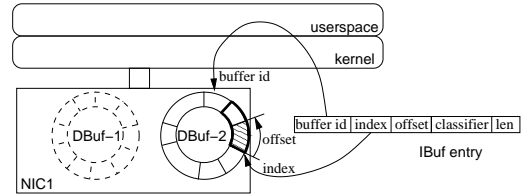


Figure 5: IBuf entry points to a byte in a DBuf on NIC<sub>1</sub>

most efficient solution. We revisit this point in Section 7.1. Finally, transformation of data (such as TCP reassembly, encryption, etc.) may be impossible to execute ‘in-place’ because of access policy or block-size limitations. In that case, we duplicate data in a shadow buffer (transparent to FEs).

**Copy Avoidance through Indirection** We use rings both for storing raw data (in DBufs) and for managing the pointers to the data. Pointers are also known as *indices* and stored in IBufs. By passing around indices instead of data streamline avoids most data copies. Indeed, DBufs are shared as much as possible, limited only by the previously noted concerns. IBufs are private to a consumer: a function or application interested in a subset of the data contained in the DBufs. For example, a mediator’s IBuf may point to the start of all frames in a videostream. Multiple IBufs may point to elements in the same DBuf and, conversely, an IBuf may have indices pointing to different DBufs. In contrast, to avoid locking, only a single producer is allowed per DBuf. Figure 4 illustrates buffer interaction. Most IBufs can be optimised away by using direct function calling between FEs. They are pertinent only on two occasions: forwarding streams across spaces and exporting them to end-users.

Figure 5 shows an index. Indices are more informative than simple pointers: as DBufs may reside in multiple address spaces, indices have to be valid globally as well. Their core is a 3-tuple consisting of a DBuf identifier, a block identifier in that ring, and an offset in the block. The first time a tuple is used within a space it is translated to a local pointer, which is cached for subsequent accesses. Indices also have a classifier field, which can group together multiple indices, such as all TCP segments belonging to the same stream. Private IBufs obviate the need for complex shared metadata structures, which require expensive locking (such as Linux’s `sk_buffs`).

IBufs can be viewed as a virtual memory layer with unconventional addressing. When accessing an IBuf its contents (indices) are not directly returned. Instead, Streamline

silently resolves the corresponding DBuf and returns a block from there. If a DBuf is not accessible from the current address space it is silently mapped in, in a manner reminiscent of page-fault handling.

**Security** Each buffer carries an individual access policy, which follows a superset of the familiar Unix file permissions: process-local, user-local, group-local and other. Note that we added to the Unix file permission an access policy per process. The reason is that sometimes users do not fully trust an application other than to read only *its* data, even if they started the application themselves.

Protection is enforced per buffer and space. This granularity is a consequence of the fact that Streamline checks permissions only on a "buffer-fault": when it is first accessed in an address space. Similar to a page-fault, a buffer-fault causes the local space to try to map in the buffer. If the buffer is found in any neighbouring spaces (not just the kernel), its access policy is compared to the current effective UID and GID. Given enough rights, the buffer is mapped in. As mapping operations are infrequent, overhead is small compared to per-call switching to kernel-mode.

Buffers encapsulate data into what we term *soft segments*: the software equivalent of memory segmentation. We would prefer to use hardware segmentation for policy enforcement, but as modern OSes favour paging we resort to combining sets of MMU-protected pages in software.

## 6.1 Uniform Interface, Specialized Buffers

The same interface is used to access any buffer anywhere (including inside the kernel), regardless of implementational or substrate peculiarities. We use the POSIX File IO interface, or **FIO** because it is familiar and convenient. FIO calls are reflected to actual implementation handlers in a manner similar to Linux's virtual filesystem switch. Reflection introduces overhead, but because specialisation can cut other, greater, costs it is worth the trouble. Because entire datarings are mapped, processes do not have to switch to kernelspace to check access permissions for every call. Hence, the API does not have to be implemented as a set of (costly) system calls. The ensuing reduction in kernel/userspace switches positively affects performance.

On the other hand, FIO is based on copy semantics, that is, `read` and `write` create private copies of blocks for the caller. These semantics are safe, but also wasteful, as they must be implemented using copying or VMM modifications such as copy-on-write. While even with the traditional `read` call Streamline offers performance improvements (as shown in Section 8), to circumvent these costs we also extend the API with `peek`, a `read`-like function that uses weak move semantics in the nomenclature of Brustoloni and Steenkiste [7]. Apart from the second argument (a double instead of a single pointer), `peek` is identical to `read`, and both functions can be used in Streamline:

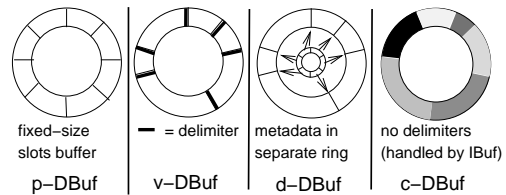


Figure 6: Different implementations of DBufs

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

**Disk access** We have integrated the page cache, the memory area responsible for caching disk blocks, in our buffering subsystem because many streams originate or terminate on disk. Server applications often loop over a `read` from disk and a `write` to a socket. The `sendfile` call, found in most operating systems, speeds up this case by bypassing the FIO. In contrast, being able to access the page cache like any other DBuf allows *splicing* [27]: generic copy-free data transfer. A `write` call compares its argument to the last block seen by `peek` or `read`. If they are the same and the write is to an IBuf (e.g., for network transmission), only the index is copied, not data.

**Memory Compression** Data buffers can be divided into two classes: continuous, or **c-DBufs** and fixed-size slotted, or **p-DBufs**. A c-DBuf accepts requests of arbitrary length. If a block overflows the buffer, it is wrapped around. p-DBufs, on the other hand, have fixed-sized slots, e.g., to hold ethernet frames. Access costs are lower for slotted buffers because they have simpler integrity checks.

Because the memory bus is one of the bottlenecks in stream processing [30], support mechanisms such as prefetching, burst transfer and caching must be used where available. Optimising these features is difficult, because cache sizes and memory latencies 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*. Usage is not the same as allocation: we focus on minimising the runtime memory footprint. Before we employ this rule we note its main exception: data alignment to hardware boundaries (cachelines, pages) benefits performance; squeezing the last bit out of each allocation is not our goal. Ring buffers know two types of waste that restrict throughput: external and internal. We discuss each below.

**Runtime Resizing** Only the part of a ring between the write- and read pointers is allocated. The rest is waste, which we call external because it lies outside the slots. Memory locality can be increased by reducing this gap. However, a buffer tuned too well to the average case will overflow during bursts. Some inefficiency is therefore unavoidable.

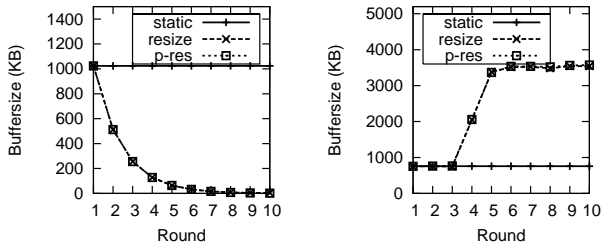


Figure 7: Resizing buffers for different rounds of resizing

Nevertheless, external waste can be limited by adapting the buffer size at runtime. To investigate this idea we built a self-resizing buffer. When the gap grows too large (or small) a new memory area is allocated to which all write operations are diverted. The old area is released as soon as all consumers have moved to the new buffer as well. Computational overhead is minimal, but memory pressure considerable during resizing.

To circumvent this issue, we also implemented a *pseudo*-resizable buffer that never changes the underlying memory area, but instead reduces the *perceived* size, similar to how a deck of cards is “cut”. Both resizing implementations trade off computational complexity for an increase in locality of reference. We compared the two to their non-resizing counterpart in Figure 7. The left figure shows how the resizable buffers neatly halve in size after each round of writes, until they stabilize around 10KB. On the right we see that performance indeed benefits. The sudden hike is attributable to an increased cache hit-rate. Not visible is the (per call) computational overhead, which was at most 10%. With compiler optimizations it was always less than 1%.

**Defaults.** Most buffers are non-resizing by default. However, a trivial strategy for resizing that is applied for many applications that tolerate loss is (1) halve the buffer size repeatedly until the buffer overflows, and (2) then double the size once. (3) If overflows are still common, double the size dynamically. (4) To prevent waste, periodically re-adjust the buffer according to steps 1-3.

**Slot Compression** As opposed to external waste, internal waste occurs within allocated slots. Buffers with fixed-size slots (**p-DBufs**) have a high percentage of internal waste, because slots are tailored to upper bounds. In case of ethernet frames slots must be at least 1514 bytes, while the majority of packets are much smaller. Minimum sized packets with the highest ratio of waste to data (up to 95% of space is unused) are quite common [9].

Internal waste is removed completely by switching to variable sized slots. In such **v-DBufs** a length field precedes each block. On a downside, seeking is considerably more expensive. Also, if no valid index into the buffer is known (e.g., because a consumer is slow and the producer doesn’t wait, a situation we will discuss shortly), reading must start

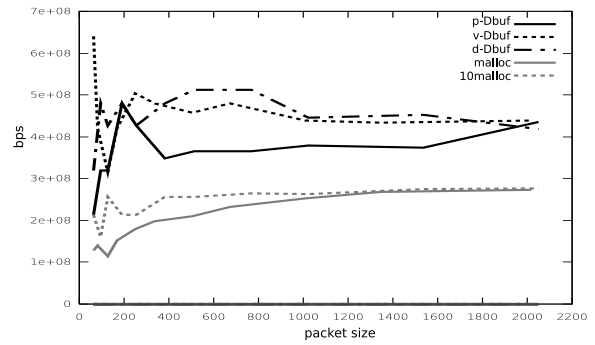


Figure 8: Comparative Ring Performance

at the head of the ring, because slots can not be identified afterwards.

Both drawbacks, seeking cost and overflow handling, can be overcome by placing the headers out-of-band, in a separate, smaller circular buffer. A small buffer fits more headers in a single cacheline, reducing seeking cost. Apart from this, such a double ring buffer – or **d-DBuf** – behaves like a **v-DBuf**.

Figure 8 compares implementation throughput. For reference, we also show allocation strategies using the `dlmalloc` general allocator (“malloc”) and a buffered version of the same that allocates 10 slots at a time (“10malloc”). Baseline ring buffer performance is about twice that of the general allocator; buffering has no effect on this trend. We also see that **p-DBufs** indeed suffer from internal waste. For maximum-sized IP packets **p-DBuf** performance is in line with the others. But as packet size shrinks, so does relative **p-DBuf** throughput: it is up to 30% slower. **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 9 shows the impact of buffer size and per-call block size on throughput for `read_stream` and `peek_stream` calls. We show only a **c-DBuf**; results were similar for other implementations. Both buffer - and call

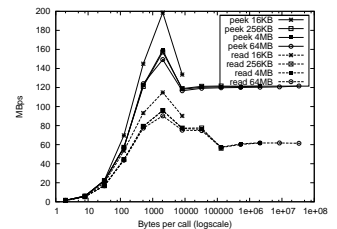


Figure 9: Impact of ring- and blocksize

sizes range from 64B to 64MB. All buffers of 16KB and smaller showed the same results. The peak where cache utilisation is maximised is clearly visible.

**Defaults.** By default, data buffers are implemented as **d-DBufs**, but monitoring applications often use **c-DBufs** for TCP streams. For file/web servers we suggest using **c-DBufs** for data and **d-DBufs** for packet headers.

**Memory Exhaustion Resolution** As rings are statically allocated, they may suffer from memory exhaustion before

the general allocator fails. For this reason they 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.

We employ three exhaustion resolution strategies: Slow, Medium and Fast Reader Preference (SRP, MRP and FRP). The methods implement different choice points on the trade-off between high speeds and low droprate. FRP and MRP are optimizations that may only be used where drops are acceptable – most notably on the network path. As its name implies, MRP is a compromise between the others. We will first introduce the extremes and then show MRP to be nearly as fast as (and to scale with) FRP, but to give the stable droprate of SRP.

SRP is commonly known as tail-drop. We call it *slow reader* because it must recalculate the position of the slowest consumer to know whether OOM holds. Doing so involves a sort over all read pointers, so SRP scales worse than linearly ( $n \log n$ ) with the number of readers. An additional drawback is that the slowest reader determines the throughput of all others.

To circumvent the calculation we developed *FRP*, which neither blocks nor drops: a write always succeeds. Instead, consumers compare their read pointer to the write pointer themselves to calculate whether an item is about to, or has been, overwritten. For this check to succeed indices into the ring 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. Also, with FRP, producers do not have to keep consumer state. Therefore an FRP buffer is easily shared read-only. But FRP is no panacea. It aggravates operation under load by allowing producers to continue to write – and thus increase load – while their data will never be read. 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, but assist with a function that can cheaply verify pointer integrity after use. The safest operation is to use `memcpy` and then verify success. But for streaming operations, or those that check for integrity at a higher level (e.g. P2P filesharing nodes) occasional data corruption at this level is preferable over recurring copy overhead.

Medium Reader Preference (*MRP*) logically sits between FRP and SRP. The model is centred around a single shared read pointer  $R$  that is updated by each consumer, which also maintains a private read pointer. The producer simply treats the buffer as tail-drop with a single read pointer  $R$ . When a consumer is scheduled it consumes  $n$  bytes and 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 the fast consumer is scheduled again. Functionally, MRP

mimics SRP except for very tardy readers. Performance-wise it resembles FRP, because slow readers cannot hold up fast readers,  $R$  update calculation is fast and synchronisation is minimal.

**Defaults.** To provide familiarity to users, we use SRP by default. However, FRP is often used across the PCI bus, as it decouples the producer from the consumers.

## 7 Implementation

Before moving on to the experimental analysis we analyse how Streamline interacts with the hardware below and applications above it.

### 7.1 Hardware Integration

Through the space interface and on-demand path construction Streamline integrates embedded and distributed hardware transparently. We ported Streamline to the IXP2xxx family of network processors as proof of concept. IXPs are asymmetric multicore processors. They combine an XS-scale CPU with a collection of special purposes RISC processors on a single chip. The RISC processors, called ‘micro-engines’ (or UEs), are not generic load-store machines. Instead, they execute code from a static small (1-4kB) instruction store, run without an operating system and lack transparent caching. On the upside, UEs can switch between (hardware) threads with zero-cycle overhead and access memory asynchronously. We have two configurations: an IXP-2400 PCI-X plug-in board and a IXDP-2850 dual processor machine that must be accessed over the network. Both have (between 3 and 10) 1-Gbit network ports.

Like VERA [23], we noticed that the XScale is not nearly powerful enough for datapath network processing. We therefore restrict the CPU to control plane operation. It talks to Streamline spaces on the host over either a PCI-X or TCP channel and carries out instantiation and buffer allocation requests on behalf of the UEs. Communication with UEs is through shared-memory.

Streamline had to tackle three complications of embedded and distributed operation: FE compilation, shipping and loading; non-uniform memory access and hardware variation – all transparently to the end-user. Our solutions to dealing with variation (word-length, byte-order) are standard; we only elaborate on the other two.

**Just-in-time Compilation, Shipping and Loading** Compiled filters, such as our pattern-recognition languages FPL3 and Ruler [35], are built at runtime, in between resource discovery and allocation. Because embedded devices are often too constrained to have an on-board compiler the code must be compiled elsewhere.

Compilation, shipping and loading are all handled automatically. The first step is a refinement of existing FE trans-

lation (see Section 3.1). An FPL3 to kernel compiler registers as `(fpl3) -> (kernel-ARCH-VERSION)`.

Because discovery can fail for many reasons and compilation is expensive it is best deferred until after discovery. A placeholder is needed to let discovery succeed without the actual code in place. We force compilers to generate agreed-upon hardware labels for the various environments (e.g., “linux-kernel-i386-2.6.17”, or “ixp1200\_uengine”). If instantiation fails, the algorithm must backtrack and select a different compiler until accommodation succeeds or all have been tried. When an FE can be accommodated and has been compiled it is shipped to the code-loading FE through Streamline’s messaging subsystem. Loading invariably occurs in a neighbouring space: an `insmod` process for kernel modules, the XScale for the UEs.

**Distributed Memory** In distributed operation, even between a host and peripheral board, the illusion of uniform shared memory is often too costly to maintain. Reads and writes across protection domains must be handled differently from local reads and writes, bit-order converted on the fly. Tailoring access methods to the application domain reduces the impact of non-uniform memory access (NUMA). Behind the FIO interface we have implemented four domain-specific buffer implementations.

**Zero-copy** access, whereby data is touched directly through remapping intuitively appears the optimal solution. However, it fails to make use of hardware support, most notably DMA. Therefore, zero-copy is restricted to protection domain crossings for which no hardware support exist, domains that physically share memory (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 – copying the buffer using hardware support is cheaper: **copy once**. This method is inefficient for the large class of applications that only access part of the stream. Waste is commonly minimised by manually limiting copying to shards of data, such as IP headers. We implemented a derived solution, **copy on demand**, which combines zero copy with copy once. In stream processing one commonly inspects a small number of bytes (a header) to decide whether the rest of the data is of interest. In this case, it is preferable to use zero-copy to access the first bytes and to incur the overhead of a full copy only 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.

Alternatively, In **cached copy** a replica allocates a large enough region to copy the entire original buffer, but fetches blocks only when requested. Performance gains are only seen when data is touched multiple times. Cache blocksize is not related to stream block size, but dependent on hardware characteristics (e.g., support for burst-traffic).

**Defaults.** Traditionally, each driver in an OS must re-

implement one of these strategies. Streamline centralises the code behind the FIO in (hot-swappable) buffer implementations. Across a shared bus the initially chosen implementation is copy-once, for network links and disk IO cached-copy, and between kernel and processes zero-copy. Orthogonal to NUMA data access selection is the choice of synchronisation. Across the PCI bus, or more generally speaking across buses with burst capabilities, frequent master switching kills performance. Here FRP performed quite well. As no state is fed back to the producer, it can keep a continuous (for all practical purposes) lock on the bus.

## 7.2 Interfaces

For backward compatibility Streamline exports well-known interfaces alongside SRL. Apart from POSIX files and pipes these are BSD sockets for network programming and PCAP for filtering. Courtesy of Streamline’s shared buffers we managed to improve the performance of the socket API substantially. Similarly, our PCAP implementation is among the fastest.

**Packet Capture with PCAP** Pcap is the most popular interface for capturing and storing live traffic. Its functionality is limited to applying a BPF expression to all incoming traffic from either a single interface, or all interfaces combined. For backward compatibility with popular tools like `tcpdump` and `ethereal` Streamline is able to export the same interface.

Streamline implements Pcap by translating the request into the SRL expression

```
(ethernet) > (bpf)
```

with the necessary options. By moving filtering to the kernel and remapping buffers, Streamline is among the fastest Pcap implementations. `PF_Ring` [14] and Phil Wood’s `mmap` patch to Pcap for Linux offer similar performance for single instances, but lack Streamline’s scalability with parallel applications and support for transparent offloading to hardware.

**BSD Sockets** Compared to `pcap`, the socket API involves more processing. Sockets abstract away tasks like multiplexing, fragmentation, reassembly, checksumming and header packing. In other words, rather than vanilla access to the rings, the sockets API provides functional processing. The single change underlying our socket calls is that they are executed in the process’s context and thus incur no kernelmode switch. Runtime performance benefits only from switch reductions in the datapath (through `recv`, `accept`, `select/poll` and `send`). `accept` is handled in userspace in the same manner as `read` and `write`: it waits on an `IBuf` with elements pointing to new flows. `recv`, `send` and related calls are even more similar to FIO: these are protocol-specific wrappers around `read` and

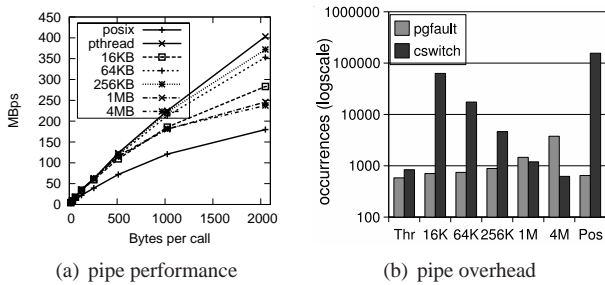


Figure 10: Evaluation of pipes

write that prepend data with headers. All these calls can be handled without the need to switch for each call because they share a ring with the kernelspace network stack.

## 8 Experiments

Where relevant we have previously supported our arguments with micro-benchmarks. We now compare our work 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 1.6.3.

**Pipes** We directly compared our Unix pipes to those within Linux. For fairness, we have not used the peek optimisation and thus copy the same amount of data. Any performance advantages witnessed must come from a reduction in context switching. To calculate an upper bound on achievable performance gain we also show results of a threaded application, which has direct access to shared memory.

Figure 10(a) shows the threaded application to outperform Linux POSIX calls by a factor of 2.5. In between are 5 different sizes of our implementation. The fastest implementation is the largest, nor smallest. Initially, performance grows with buffer size. This is to be expected as, the number of context switches drops when calls are less likely to block. Unfortunately, page-faults start affecting performance when the TLB runs out of entries. A Pentium 4 has 64 data TLB entries, corresponding to 256KB of virtual memory. Peak throughput occurs with rings sized between 64 and 256KB. Within this range the number of context switches grows minimally, while we do not yet witness TLB thrashing. The same holds for the threaded application, even though we only show its optimal result for clarity.

Orthogonal to ring size is function call size. For calls with small blocks CPU processing is the main overhead. Here context switch reduction is most beneficial. When the bottleneck shifts to memory, throughput starts to scale linearly. Figure 10(b) plots the context switches and page-faults independently (aggregated over all block sizes). As expected, context switch count diminishes super-linearly until it sta-

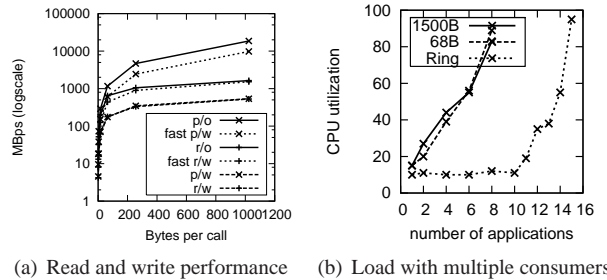


Figure 11: Reading, writing and multiple consumers

bilises at 4MB. Offsetting this advantage, from 256KB onwards page-fault occurrences grows rapidly.

**Copy Avoidance** The pipe test showed that our rings transport data more efficiently than Linux pipes. We now take a closer look at some optimisations and compare performance of peek to that of read, with and without the fast splice optimisation. Figure 11(a) shows the performance of reading and writing data for a file server application, where data is read from the page cache (which is filled automatically by means of prefetching) and written to a network transmission 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 only metadata is touched. However, read-only access is not useful and only shown as upper bound on the performance. Barely slower is fast peek/write, which combines peek with the fast splice optimisation. This, too, does not actually touch data, but must write 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: throughput drops again, to a third. Writing must be the main bottleneck as peek/write and read/write are equally fast, while in other cases peek clearly outperforms read.

**Parallel tasks with PCAP** Increasingly, auxiliary processes are attached to existing datastreams, such as virus scanners, IDSs, profilers. While current systems incur copies for each application, our rings are capable of sharing buffers *horizontally* between processes as well.

Figure 11(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 Streamline’s PCAP interface. We measure throughput for a modest datastream: 100 MBit of large sized packets per second (i.e. 7500 pps). This allows us to investigate scalability. Our version is about 25% more efficient than the stock implementation for a single application. More interesting savings occur when we run applications in paral-

lel. Whereas standard tcpdump scales linearly with the number of packets, our version incurs no significant overhead for up to 9 applications. Streamline moves most processing to the kernel, where it merges the FEs. Furthermore, we have configured tcpdump to output only minimal information.

With more than 10 applications thrashing occurs. Inspecting the number of voluntary and forced context switches independently shows that, although involuntary switching increases for each extra application, thrashing does not occur until the number of *voluntary* switches starts decreasing. This indicates that a process fails to process within its timeslot, but needs to defer, starting a snowball effect.

Tcpdump is expected to be memory bound, but according to Figure 11(b) the minimal and maximal capture length versions have similar overhead. 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 Streamline 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, the common case in non-monitoring applications.

**Socket Processing** Our socket interface is layered on top of two SRL expressions, one for input and one for output. For the moment we use Linux networking routines to achieve driver independence, but this comes at a cost. In both cases we incur one unnecessary copy. Like traditional sockets, we also incur an overhead in the interface itself for reception. One the reception path this limits throughput to roughly equivalent to Linux. Transmission is even about twice as slow as Linux without further optimization. Comparing these numbers with PCAP indicates that they can be improved.

**Hardware Support** We have applied Streamline as an embedded intrusion prevention system (IPS) on the IXP2400. The IPS consisted of TCP reassembly, regular expression matching and protocol deconstruction FEs. Despite the substantial computational overhead, the system achieved a throughput of just below 1 Gbps even in the worst case [35].

## 9 Conclusion

By constructing all communication paths automatically, on demand, Streamline takes an extreme position in the OS design space. It improves performance by up to an order of magnitude compared to conservative approaches. Separating logic into distinct processing-, buffering- and control planes further opens up paths for optimisation, which

incidentally shows that structured OS development and efficiency are not mutually exclusive. Finally, Streamline demonstrates that extensibility does not have to overwhelm the end-user or application developer. Through a concise language and a handful of heuristics it relieves them of meddling with technical detail.

Streamline is practical software that can be directly applied to networking tasks, such as intrusion prevention, multi-layer routing and media streaming. The architecture facilitates integration of next generation heterogeneous hardware (Cell Processors, integrated GPUs) in a single coherent system. Streamline's immediate impact consists of optimisation of the datapath from which we removed as much overhead as possible (allocation, copying, context switching, signalling).

## References

- [1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proc. of CIDR 2005*, Asilomar, CA, January 2005.
- [2] B. Ahlgren, M. Bjoerkman, and P. Gunningberg. The applicability of integrated layer processing. *IEEE JSAC*, 16(3):317–331, Apr. 1998.
- [3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *OSDI*, pages 115–123, 1994.
- [4] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. Silk: Scout paths in the linux kernel. Technical report, Uppsala University, 2002.
- [5] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.
- [6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the spin operating system. In *SOSP*, pages 267–284, Copper Mountain, Colorado, 1995.
- [7] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *Operating Systems Design and Implementation*, pages 277–291, 1996.
- [8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

- [9] K. Claffy, G. J. Miller, and K. Thompson. The nature of the beast: recent traffic measurements [...]. In *Proc. of INET'98*, 1998.
- [10] D. Clark. The structuring of systems using upcalls. In *SOSP '85*, pages 171–180, New York, NY, USA, 1985. ACM Press.
- [11] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90*, Philadelphia, PA, USA, September 1990.
- [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] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *Knowledge Discovery and Data Mining*, pages 9–17, 2000.
- [14] L. Deri. Improving passive packet capture:beyond device polling. <http://luca.ntop.org/>, 2004.
- [15] P. Druschel, M. B. Abbott, M. A. Pagals, and L. L. Peterson. Network subsystems design. *IEEE Network*, 7(4):8–17, 1993.
- [16] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*, pages 189–202, Ashville,NC, 1993.
- [17] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM'96*, pages 53–59, 1996.
- [18] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: a safe programmable and integrated network environment. In *ACM SIGOPS European workshop*, pages 7–12, New York, NY, USA, 1998.
- [19] R. Govindan and D. P. Anderson. Scheduling and ipc mechanisms for continuous media. In *SOSP*, pages 68–80. ACM SIGOPS, 1991.
- [20] T. S. Group. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.
- [21] 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.
- [22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of Eurosys'07*, 2007.
- [23] S. Karlin and L. Peterson. Vera: an extensible router architecture. *Computer Networks (Amsterdam, Netherlands: 1999)*, 38(3):277–293, 2002.
- [24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
- [25] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Computer Communication Review, ACM SIGCOMM, volume 28, number 4*, Vancouver, Canada, Oct. 1998.
- [26] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX conference*, San Diego, Ca., Jan. 1993.
- [27] L. McVoy. The splice I/O model. [www.bitmover.com/lm/papers/splice.ps](http://www.bitmover.com/lm/papers/splice.ps), 1998.
- [28] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*, 1994.
- [29] 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.
- [30] J. Paisley and J. Sventek. Real-time detection of grid bulk transfer traffic. In *Proc. of IEEE/IFIP NOMS'06*, 2006.
- [31] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.
- [32] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of USENIX 98*, pages 13–24, 1998.
- [33] M. Welsh, D. E. Culler, and E. A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.
- [34] X. Reference removed for blind review. 2004.
- [35] X. Reference removed for blind review. 2006.