Application-tailored I/O with Streamline

WILLEM DE BRUIJN, HERBERT BOS and HENRI BAL, Vrije Universiteit Amsterdam

Streamline is a stream-based OS communication subsystem that spans from peripheral hardware to userspace processes. It improves performance of I/O-bound applications (such as webservers and streaming media applications) by constructing tailor-made I/O paths through the operating system for each application at runtime. Path optimization removes unnecessary copying, context switching and cache replacement and integrates specialized hardware. Streamline automates optimization and only presents users a clear, concise job control language based on Unix pipelines. For backward compatibility Streamline also presents well known files, pipes and sockets abstractions. Observed throughput improvement over Linux 2.6.24 for networking applications is up to 30-fold, but two-fold is more typical.

Categories and Subject Descriptors: D.4.4 [**Software**]: Operating Systems—*Communications Management*; D.4.8 [**Software**]: Operating Systems—*Performance*

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: I/O buffering, streams and filters, zero-copy

1. INTRODUCTION

The bottleneck in system software has moved from the CPU to the memory system, especially for I/O-intensive tasks such as networking [1995]. Operating systems have not structurally changed to reflect this reality, with the result that architectural decisions made in the past hinder applications today. They waste CPU cycles copying data between kernel subsystems and across memory protection boundaries. They waste cycles switching tasks too frequently. To make matters worse, they waste cycles refreshing caches as a result of all this copying and context switching. An application binary interface (ABI) at the abstraction level of Posix calls incurs many mode switches between userspace and kernel mode by handling packets one at a time; multi-user OS access control imposes copy semantics across memory protection domains even on dedicated (i.e., single user) servers. These inefficiencies result from fundamental architecture choices and can only be resolved through comprehensive OS restructuring. Failure to resolve the issues systematically has led to application-specific solutions, such as disk caches duplicated in userspace (in server-side script engines) or kernelspace application servers. This road is far from ideal, as it increases code complexity, memory- and CPU utilization, and reduces robustness.

A second operating system I/O obstacle, and one that is rapidly gaining importance, is lack of support for heterogeneous hardware designs, such as the Cell or AMD Fusion. The Cell may be the first asymmetric multicore processor in wide use, but all

This work was supported in part by the EU through the projects SysSec (FP7-ICT-257007), and i-Code (funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission Directorate-General for Home Affairs). Author's address: W. de Bruijn, Computer Systems, Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

major manufacturers have plans to integrate specialized logic such as programmable graphics/SIMD coprocessors, network interfaces (NICs), encryption modules and even FPGAs on-chip. As specialized resources can process up to orders of magnitude faster than equivalent software running on a CPU, these must be employed universally.

The operating system is capable of safely and fairly multiplexing resources among all tasks, but traditional monolithic systems impose strict functional boundaries between application, kernel and device and limit hardware support to a handful of hardwired operations, such as TCP checksumming. Manycore computer systems offer more flexible layouts, including space multiplexing and heterogeneous cores, but conversely, need new programming models to economically define malleable applications that scale with hardware resources and workload.

An *I/O architecture* is the communication fabric linking applications, OS kernel and hardware, that extends from library interfaces in userspace down to peripheral devices. It crosscuts the classical OS layering. The present I/O architecture not only impedes performance on conventional hardware, it also obstructs the use of heterogeneous hardware. In this paper we present Streamline, an I/O architecture for commodity operating systems that avoids common I/O bottlenecks and enables clean integration of arbitrary hardware. Streamline specializes I/O logic on-demand to match application profiles and hardware configurations, a design that we term *application-tailored I/O*. To achieve this, it has the following distinctive characteristics:

(1) a buffer management system for I/O that avoids common copy, context switch and cache miss overhead through shared memory transport and indirection.
(2) a dataplane that bypasses bottlenecks and integrates all hardware by selecting suitable implementations of logic on-demand to form *I/O paths*: graphs of processing and buffering elements.
(3) a control system that automatically translates application requests expressed as abstract Unix-like pipelines into I/O path implementations tailored to the application profile and local hardware characteristics.

On top of this we have engineered legacy I/O interfaces to allow direct comparison with Linux. The contribution of this paper is to show that this architecture:

I Increases Unix primitive throughput 2x to 30x over standard Linux.
II Increases legacy application throughput up to 4x.
III Increases throughput further when modified call semantics are allowed.
IV Enables intrinsically efficient and portable native applications.

This paper is organized as follows: we start by presenting the application domain for Streamline in Section 2. Section 3 introduces the architecture and Section 4 presents the interface. The following three sections each detail one of the main components: buffering (Section 5), processing (Section 6) and control (Section 7). Section 8 discusses security aspects; Section 9 evaluates the system quantitatively, the next discusses related work and Section 11 draws conclusions.

2. APPLICATION DOMAIN

This paper identifies and removes structural bottlenecks in operating system I/O. To limit the problem space, we focus on network applications. Disk, graphics and audio processing are discussed in as far as they affect network applications. Disk I/O, for instance, is a critical factor in web serving. We target dedicated workstations and servers (as opposed to multi-user time-sharing systems), because these are predominant today. While Streamline offers Unix-equivalent isolation, we demonstrate that performance can increase further when a reduction in isolation is accepted. Our primary measure of performance (and hence success) is throughput.

It is harder to demonstrate utility of a new I/O architecture by improving existing application throughput than by handcrafting programs to make use of all unique features. Besides evaluating individual contributions to show their merit in vitro, pipes to show real Unix primitive improvement, and native applications to demonstrate novel features, we measure performance of several mature legacy applications, to demonstrate that Streamline is useful even for applications not written with its design in mind.

2.1. Representative Legacy Applications

Because no single task can expose all system weaknesses, we have compiled an application set for benchmarking, each element of which exemplifies a common I/O processing profile. As our current TCP implementation is incomplete (and TCP itself is not our prime concern), we selected a variety of UDP applications.

DNS server. DNS servers represent critical applications that must scale to high request rates, but incur only minimal memory- and disk I/O. This characteristic makes them well-suited to identify context switch and per-packet computational overhead. We benchmark the industry standard daemon `bind`.

Streaming video client. Video streaming also scales to high rates, but with large per-packet payloads. This task demonstrates memory-bound processing. We use the popular `mplayer` application for our benchmarks and avoid stressing the CPU by disabling codec processing.

Traffic analyzer. Traffic analyzers, such as intrusion detection systems, connection trackers and malware blockers, are secondary tasks that attach to existing I/O paths. They show the cost of data duplication in an I/O architecture. We attach the well-known `tcpdump` analyzer to a moderate datastream to observe multitask performance degradation.

Disk duplicator. A common pattern is to stream data from the disk (or disk cache) to the network. We isolate the performance benefit of zerocopy transfer from the cache by copying data using the popular Unix `dd` disk duplication utility.

2.2. Bottlenecks

On monolithic operating systems such as Linux, streaming I/O applications encounter one or more of the common bottlenecks presented in Figure 1. Transport overhead accrues where data is forwarded, at the crossings between hard- and software compartments. Computation issues can occur anywhere; these are the result of a poor match of application to available hardware. We now discuss the six bottlenecks.

1. System Calls. Commonly, processes communicate with the kernel through system calls that require a mode-transition and copy operation for each block.

2. IPC. System call overhead affects inter process communication (IPC) most. Traffic between applications is copied twice and per-call block size is constrained (often to 4 KB), causing frequent task-switching.

3. Group Communication. Multiprocess access to the same data is seen in group communication (which subsumes 2-party IPC) and when auxiliary tasks such as traffic monitors are enabled. As in the IPC case, access to shared data requires a copy for each process and frequent task-switching.

4. Kernel Subsystems. Between kernel subsystems copying is required when interfaces are incompatible. A classic example is having to copy between the disk cache and network queues while pinning of memory pages suffices in principle.

5. Direct I/O. Data traverses the kernel even when it performs no operation. High-speed devices (e.g., DAG cards [Cleary et al. 2000]) present libraries that bypass this bottleneck, but these require superuser privileges and exclusive device

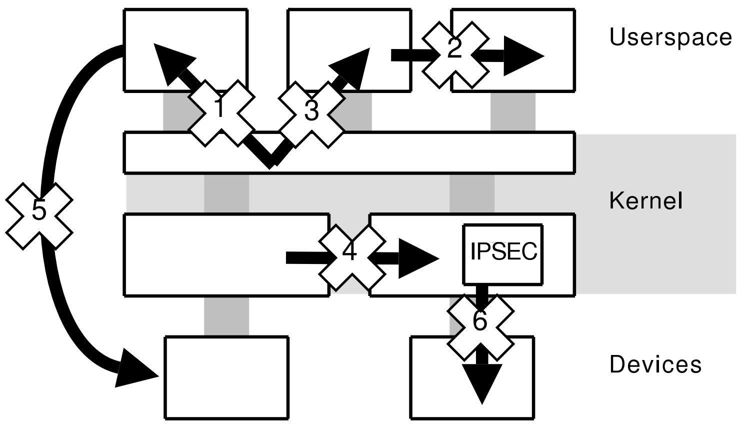


Fig. 1. I/O Bottlenecks in a monolithic OS—the numbers are explained in the text in Sect. 2.2

access and they replace generic I/O primitives with vendor-specific APIs. An OS approach combines standard kernel control for resource multiplexing and device configuration with generic kernel bypass interfaces on the datapath.

6. Fixed Logic. Applications sometimes encounter the above bottlenecks unnecessarily, because OSes force all to structure I/O logic the same way. A fileserver can save two copies by moving fast-path logic to the kernel; a DNS daemon can reduce latency by bypassing the kernel completely.

In Section 9 we return to the presented bottlenecks and applications and analyze Streamline's effect on them. First, we present its architecture in detail, starting in the next section with a overview.

3. ARCHITECTURE

Application-tailored I/O as espoused by Streamline avoids common I/O bottlenecks by reconfiguring datapath logic at application load time to match workload and exploit special-purpose hardware. In general, the two extremes in coping with heterogeneous hardware and software configurations are (a) not to deal with it at all (static code), and (b) fully recompile all the code with specific optimizations for each specific configuration. Streamline implements a mid-way point between static code and full recompilation, because both are impractical. The first cannot anticipate all computer architectures and applications. The second requires a single tool-chain capable of programming all available devices on each end host. Instead, Streamline constructs application-tailored *I/O paths* at runtime from sets of precompiled processing and buffering elements. It avoids bottlenecks by optimizing the mapping of applications onto the physical computer architecture: a *tailoring* algorithm selects the set of elements that (1) satisfies the application, (2) maximizes the use of specialized hardware and (3) minimizes data movement, in that order. The Streamline architecture consists of three components: processing, buffering and control.

Processing. Automatic path optimization requires a simple application model, for which purpose Streamline reuses the well known streams and filters model [1984]. It refines the implementation to avoid unnecessary cost from context switching, copying and cache misses. Specifically, it moves processing close to the data and minimizes data copying and task switching between stages.

The streams and filters model is appropriate for streaming I/O in principle, but the most widely used implementation, Unix pipelines, introduces considerable data movement and task switching cost. As a result, Unix pipes are much loved, but little used when performance matters. An implementation based on low level `fork()` and `pipe()` calls lacks a global application view and therefore fails to exploit modern computer architecture opportunities such as cache-aware or gang scheduling.

Besides scheduling, Unix pipelines impose considerable I/O cost. Processes spend many cycles reading from and writing to pipes, a source of copying and task switching. This cost would be unavoidable if filters modify data considerably, but many only want to read, group, split, discard or append it, all of which can be implemented more efficiently. For instance, in networking, many filters touch data only for reading, such as in protocol demultiplexing, or to make minor changes, such as in header stripping. Because data passing between filters is one of the most often executed parts of the system, keeping this task cheap is essential.

Streamline incorporates existing Unix processes as filters, but also more efficient callback functions that support signal moderation and shared memory. Unlike a Unix OS scheduler, the Streamline runtime system uses its end-to-end view of pipelines to optimize filter placement, signaling and data movement across user processes, kernel tasks and devices. Filters inform processing by annotating data with metadata, in the form of per-block *classification values*. For instance, an MPEG video decoder can announce the frametype of a data block to downstream filters. A TCP reassembler identifies the particular TCP stream that each segment corresponds to.

Buffering. Because the impact of buffering on overall performance continues to grow as memory latency falls behind CPU speed increases [Wulf and McKee 1995], buffer management is a critical factor in I/O application performance, While the dataplane presents a view of streams and filters, underneath data moves using a different shared memory model to minimize copying and maximize cache hitrate. Reconfigurable systems require flexible buffer management that can route data to where it is needed. We present a buffer management system (BMS) that spans across software tasks and peripheral hardware and that is designed for throughput as follows: all live data is kept in coarse-grain ring buffers, buffers are shared long-term between protection domains and data transformation is replaced with updates to metadata structures.

Control. A pipeline easily comprises tens of buffering and filtering elements. This is the basis for *application tailoring*: optimization at application load-time of I/O paths by selecting filter and buffer implementations that maximize throughput. Constructing an optimal I/O path configuration for each application type and hardware architecture is a long, repetitive and error-prone task, involving resource discovery, -selection and -allocation. To minimize code duplication and prevent suboptimal configurations, this task is automated instead of passed on to application developers. Streamline only presents its users with a simple and familiar declarative interface that resembles Unix shell expressions.

Implementation. We have implemented Streamline as a combination of a Linux kernel module, a userspace library and a set of device drivers that completely replaces native I/O in Linux 2.6 (for x86 and AMD64 targets). Streamline is open source[1] software and has been used to build real performance-critical network applications, such as an application-layer firewall and a 10Gb cryptographic token-based switch. Streamline has also been ported to userspace Solaris and Intel IXP 1200 and 2x00 network processors.

[1]Available from `http://netstreamline.org/` under a mixed BSD/LGPL license.

The three components of buffering, processing and control form the main Streamline I/O architecture. The next section introduces Streamline through it interfaces. The following three sections each discuss one major architectural component.

4. INTERFACES

Streamline reuses proven interfaces where possible. We base our syntax on Unix I/O [Ritchie 1984] and only deviate when functional or performance constraints demand it. More important than syntax is that it follows the design principles of Unix: in particular, that "everything is a file", i.e., that all resources live in the same filepath namespace and expose the same file interface. In Unix, several resources actually do not conform, such as semaphores and network channels. Also, much high performance I/O is pushed to kernel and device pipelines (network, video, audio and graphics processing) that lack access to the Unix interfaces – most likely in part because Unix I/O is considered too inefficient. Streamline opens up these kernel streams through a virtual filesystem, to render all system I/O programmable with existing tools and so enable shell programming of high-rate tasks. It applies the Unix shell language to network application programming, which it extends with parallelism and (Boolean) filtering support. To be able to offload work, it applies the same shell job control to kernel and peripheral tasks. The single system-wide interface ensures that the buffering and processing enhancements of the next two chapters apply universally and that the control plane can optimize code placement throughout the stack without interface compatibility concerns.

4.1. Stream Filesystem

Streamline renders kernel streams and pipelines accessible to Unix applications and shells through PipesFS [de Bruijn and Bos 2008b]: a filesystem that presents pipelines as a directory hierarchy where each directory encapsulates a filter and nesting symbolizes data flow from parent to child. The output of each filter can be tapped from a Unix pipe node in the directory. The filesystem interface serves a dual purpose. One, it exposes all active I/O streams in the kernel to user processes, opening them up to sophisticated monitoring and transformation applications, including existing tools such as `grep` and even shell scripts. Second, through directory operations such as `mkdir` and `slink`, it exposes a low-level pipeline construction API through which applications can modify kernel logic at runtime. A filesystem interface makes it trivial to, for instance, log all webserver requests to a compressed archive. For PipesFS mounted at `/pipes` the shell job

```
cat /pipes/rx/ip/dport/80/untcp/http/get/all | gzip > log.gz
```

suffices. The `gzip` application reads data produced by an in-kernel *get* request filter from that filter's Unix pipe `all`. The filter first acquired data from another filter, one for *http* traffic, which received it from yet a lower layer, etcetera. Modifying a pipeline, for instance to insert a layer 7 protocol filter when a vulnerability in the webserver is discovered, is as simple. The filter does not have to run in the vulnerable kernel: a userspace program, which can be as common as `grep` or `sed`, can be placed in between two kernel components. We return to this example and discuss its performance implications after introducing the pertinent filesystem components.

Pipeline Control. Each directory in PipesFS represents an active Streamline filter running in the kernel. Users can insert, remove and mirror directories with the `mkdir`, `rmdir` and `slink` Linux system calls. Conceptually, data flows down the tree from parent to child directories. Directory creation with `mkdir` constructs a new filter in the kernel and connects its input to the output of its parent. Mirroring with `slink` is es-

```
if ((fd = open("/dev/term/a", O_RDWR)) < 0) {
  perror("open failed");
  exit(1);
}
if (ioctl(fd, I_PUSH, "chconv") < 0) {
  perror("ioctl I_PUSH failed");
  exit(2);
}
```

Fig. 2. Inserting a STREAMS module in Solaris 10

sential to define non-linear graphs. Creation of a symbolic link in a new directory makes that directory an additional parent of the link destination. Once created, filters can be freely moved and copied. Moving a filter (executing the `rename` system call on the directory node) severs the input stream and attaches to the new parent's output stream.

Stream access. Each directory holds a pipe node that gives access to its output stream. Applications can read a duplicate of a kernel stream by opening this pipe node; writing to the pipe inserts data into the kernel stream. To transform a stream, an application severs the original stream between parent and child by moving the child directory to the root level, reads from the former parent's output pipe and writes to the child's input pipe. For instance, to implement a layer 7 (i.e., deep inspection) protocol filter, traditionally, all network data must be intercepted in the kernel and parsed up to the protocol level. With PipesFS, not only can we move this processing out of the kernel, we can easily attach at any stage in the network stack. Let's say we want to drop all HTTP requests containing the unsafe double-dot (`..`) notation before they reach our server. The following shell script achieves this (albeit crudely).

```
mkdir /pipes/httpclean
mv /pipes/[...]/http/get /pipes/httpclean/
cat /pipes/[...]/http/all | grep -v '..' > /pipes/httpclean/all
```

It creates a new directory at the filesystem root, moves the HTTP GET filter in here, then manually reads the HTTP packets from the original parent, scans them for the double dot (with grep, which drops matches when `-v` is given) and writes the results to the outgoing filter's parent's pipe.

Communication between kernel and userspace causes task-switching, cache flushing and copying. To reach high rates, PipesFS relies on selective buffering between filters and shared memory streams between kernel and application. PipesFS buffers data only when a process actively listens on a tap. As a result, the existence of kernel taps is in itself not a source of overhead: by selectively opening streams users can trade off overhead and programmability. Section 5 presents the shared memory channels on which PipesFS relies for fast kernel I/O access.

4.2. Pipeline Job Management

For job control of adaptive applications, Streamline builds a declarative shell language on top of PipesFS. Many streaming I/O systems expose a graph composition language similar to PipesFS. Figure 2 reproduces a code snippet from the Solaris 10 STREAMS programming guide [Sun Microsystems 2005] that shows how to insert a filter into the kernel. The snippet inadvertently demonstrates how verbose and error prone such explicit programming of streams and filters is. A declarative language is more concise and robust, because it automates such boilerplate and can offer robust transactional semantics. Critically, the high level of abstraction also makes end-to-end optimization of I/O paths possible.

Streamline presents a pipeline interface and borrows its syntax from Unix shells. It goes beyond existing shell programming by applying the language also to protected kernel and device operations. For instance, the pipeline

```
rx | tcp | http | files
```

reassembles all incoming TCP streams, selects connections that contain HTTP requests, extracts these requests and saves them to individual files. To be able to express common networking tasks, such as load balancing, protocol demultiplexing and connection handling, Streamline adds two base features to the pipeline: parallelism and conditional execution. On top of these, it then constructs optional higher level control flow, such as looping, Boolean expression and connection handling, without changing the basic pipeline and at no additional runtime cost.

The plus symbol ('+') identifies parallel compositions, as opposed to the sequential compositions of the pipeline ('`|`'). Parentheses may be used to override default operator precedence. The following request, for instance, defines a firewall that only accepts three protocols.

```
tcp | http + ssh + rtsp | inspect
```

The '+' operator can represent split-join parallelism, but another notation is required to extend the language to arbitrary digraphs. Arcs that flow to earlier defined parts of the graph to form cycles are expressed by marking multiple nodes in the expression as duplicates. For example, the expression

```
a --name=X | b | a --name=X
```

maps onto a basic cycle.

Streamline adds arc constraints to pipelines to be able to filter traffic based on the classification value returned by filters in the pipeline. A default pipe forwards all classes of data except for class zero. More interesting constraints are set through annotation of the pipe symbol. For example, traffic forwarding is restricted to port 22 with: '`dport |22 ssh`'. Selection is inverted by inserting an exclamation mark. On a webserver device that only accepts port 80 traffic, all other data is reflected to an intrusion detection system using `dport |!80 ids`. For ease of use, Streamline also understands ranges ('`dport |22:23 log`') and filter-defined names ('`dport |ssh:telnet log`').

Boolean Selection. Often, users want to express filtering as the intersection, union or inversion of streams. Streamline introduces notation for Boolean selection and implements these on top of class-based filtering, i.e., without changes to the runtime system. We append another plus symbol ('+') for split-join parallelism with duplicate removal – in other words, as the union of a set of streams which is equivalent to a logical *Or* operation. Analogously, we denote intersection, or the logical *And*, by adding a multiplication symbol ('+*') and negation with the exclamation mark ('!'). All three can be seen in the example

```
rx ++ tx | ip | mpeg +* !http
```

Here, all network traffic is sent to two protocol filters; only video over anything but HTTP is kept.

Connection Handling. For applications with many short-lived connections, the quintessential example of which is a webserver, addition and deletion of pipes at runtime for each connection is too expensive. Connections are more efficiently handled by multiplexing them over a single long-lived stream. For example, the expression

```
rx | ip | tcp | pcap | compress | files
```

encodes a request that filters IP traffic, reconstructs independent TCP streams and stores each to an individual compressed tracefile.

Connection multiplexing is difficult to implement in a generic fashion, because it requires bookkeeping (and hence state) and because this state must be shared between otherwise independent operations across protection boundaries. Layer-3 routers, including those based on streams such as Click [Kohler et al. 2000], have no need for connection multiplexing because they act only on discrete packets. Other network stacks differentiate streams internally within subsystems, such as within the TCP protocol handler, but lack a common interface to share this information with other components. Such solutions are task-specific and localized, causing multiple code blocks to reimplement the same session bookkeeping and to execute the same logic to assign blocks to sessions. In Streamline, instead, filters can agree to use the classifier to discern among logically independent streams. Then, only the first filter in a pipeline has to assign a filter to a session; all successive filters refer to the classifier value to group blocks into sessions. The TCP stream reassembler, for instance, sets the tag to differentiate among many independent TCP streams over a single pipe.

4.3. Legacy Interfaces

To support legacy network applications, Streamline implements the popular network sockets and `pcap` packet capture interfaces. Both implementations are layered on top of pipelines, to automatically benefit from fast buffers and I/O path optimization.

Packet Capture. Packet capture is increasingly common, for instance in auditing and intrusion detection. The most established packet capture interface, simply known as *pcap*, applies a Berkeley Packet Filter [McCanne and Jacobson 1993] to all layer 2 network traffic on a host and sends matching frames to a userspace process. Traditionally, and in Linux today, each frame requires a mode-transition and a copy for each interested process. Shared ring-buffers implement a more efficient capture solution, because they remove all runtime copying and virtual memory mapping overhead (as we will show quantitatively in Section 9.2). Streamline directly maps all layer 2 data to all interested applications, together with the metadata streams that holds the result of a BPF filter (this mechanism is discussed in Section 5). It implements the task as a pipeline, so that the BPF filter is automatically offloaded or replaced by a more efficient implementation when available. One such implementation that Streamline carries is FPL, which compiles to native code, currently with x86 and Intel IXP network processor targets [Bos et al. 2004]. The following request takes all network sources, applies BPF and exports the stream to userspace.

```
rx + tx | bpf "tcp" | user
```

Sockets. Sockets encapsulate common network protocol processing. A socket performs multiplexing, fragmentation, stream reassembly, checksumming and header generation. As sockets handle both ingress and egress traffic, each socket is implemented as two pipelines. For instance, for a DNS server, the reception pipeline is

```
rx | udp | dport |53 socket export=yes name=rxport_53
```

It reads all arriving network data, filters out UDP traffic to port 53 and attaches it to a socket endpoint. This last filter saves all contents to a buffer named `rxport_53`. The socket calls (`recv`, etc.) operate directly on this buffer. The transmission pipeline is shorter:

```
buffer txport_53 |all tx
```

The first filter sets up a buffer for transmission, `txport_53`, and begins to listen on all writes to this buffer, which it forwards data to the transmission filter.

5. BUFFERING

A buffer management system (BMS) controls the movement of data through a computer system. Its design determines the number of copies, data-cache misses and task switches per block. To maximize end-to-end throughput, these technical buffering details must be managed centrally, concealed from individual data clients, such as applications and filters. Therefore, the Streamline BMS[2] presents clients only with an idealized view of streams, as sliding windows over principally unbounded sequences of blocks.

In this section we explain how Streamline implements this BMS efficiently, i.e., with a focus on copy -, context switch -, and cache miss avoidance. We introduce a copy and switch avoiding implementation of single streams based on ring buffers and scale this to a practical multi-ring system that uses indirection to avoid copying between buffers. Additionally, we demonstrate how buffers can inflate and deflate to optimize signaling rate and cache utilization and how rings form a basis for zero-copy communication channels between applications and high-speed I/O devices.

5.1. Streams and buffers

Because it is appropriate for sequential access and well known, streams export the classic Unix file interface (`open()`, `read()`, etc.) Traditionally, this interface is implemented as part of the ABI, but Streamline makes it available in all spaces and without a mode transition: all calls are local *function* calls that operate on locally accessible memory buffers (unless dictated otherwise by data access policy). Buffers are implemented as large contiguous memory regions capable of holding many blocks. To transport data across memory protection domains with minimal overhead, Streamline shares these regions among domains. Previous work has shown that modifying virtual memory mapping is cheaper than copying [Pasquale et al. 1994]. We increase these savings by reusing the same mappings for the duration of an I/O path. We will demonstrate that the reduction in required context-switches improves small block performance.

Pipes. Streams can be viewed as Unix pipes, as both implement volatile streams behind the Unix file interface. Because pipes map directly onto buffers, all optimizations presented for buffers are also available to pipes. In particular, this means that Streamline offers pipes based on shared memory and local function calls, backed by memory regions scaled to cache size. This model weakens isolation guarantees; if necessary, buffers also expose a legacy system call interface.

To support concurrent clients in parallel parts of the application graph, Streamline introduces *multiway* pipes: group communication channels that multiplex all input onto all output channels. The function call

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

creates a single producer descriptor and $num - 1$ consumer descriptors, each with a private file offset. We expect that group communication primitives will become increasingly important as on-chip processor parallelism increases. Multiway pipes present a simple interface for master-worker style communication.

Copy Avoidance. One performance drawback of Unix I/O is that it implements expensive copy semantics, that is, `read` and `write` create private copies of blocks for the caller. To avoid this cost, we extend the API with `peek(int, char **, int)`, a `read`-like function that uses weak move semantics [Brustoloni and Steenkiste 1996]. With

[2]Presented individually as Beltway Buffers [de Bruijn and Bos 2008a]

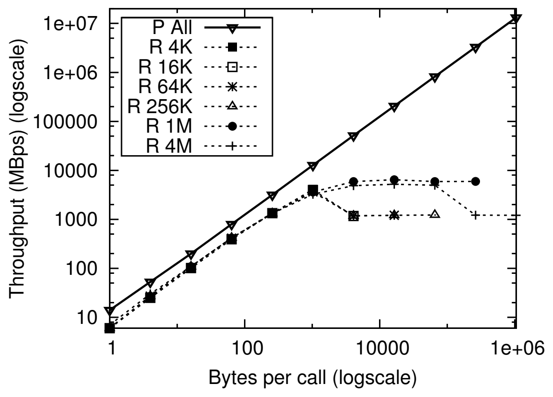


Fig. 3. Copy avoidance with `peek`. For clarity, peek ('P') results are collapsed, because they all overlap.

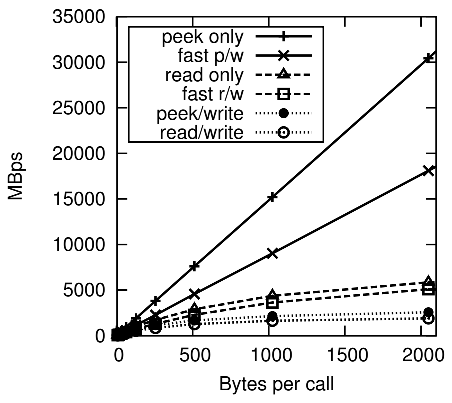


Fig. 4. Throughput with splicing ("fast")

`peek`, a client receives a direct pointer into the stream. The `read` call, then, is nothing more than a wrapper around `peek`, `memcpy` and exception handling logic. Figure 3 shows the gains obtained by switching from a copy-based `read` (R) to an indirect `peek` (P) call. The figure plots throughput for various DBuf sizes at increasing call sizes (size of the application buffer passed in `read` and `write` calls). As expected, peek throughput scales linearly for all buffers, as it is purely computational. Read throughput, on the other hand, experiences `memcpy` overhead. Even for the smallest packets, it is about one third slower than peek. Where possible (internally and for error-tolerant applications such as P2P clients), Streamline uses the `peek` call.

5.2. Ring buffers

Traditionally, operating systems allocate blocks on-demand and use pointer queues to group blocks into streams. In contrast, we build static data rings, or *DBufs* from large memory regions. A single *shared* ring has previously been shown to reduce copying cost between the kernel and userspace processes [Govindan and Anderson 1991; Bos et al. 2004]. Static, shared rings hold a number of advantages over I/O based on dynamically allocated blocks: they amortize allocation and virtual memory management operations over the lifetime of streams and render sequential access cheap within streams because blocks are ordered in memory. These advantages are offset by two challenges. First, they trade memory utilization for speed. As memory density grows faster than bandwidth, trading off space for speed is increasingly appropriate. Moreover, we will show that memory pressure can be curtailed. Second, rings are coarse-grain structures that lack per-block policy enforcement. With few policy groups and devices (the common case), these issues can be resolved by switching to a multi-ring architecture. We observed three obstacles to moving to a ring-based architecture, all of which are resolved by splitting data into a handful of rings:

Security. Processes must be isolated from one another to guarantee data privacy and correctness. Each Streamline buffer carries an access control policy that follows familiar Unix file permissions: user, group and other, for both reading and writing. As with files, with multiple rings, arbitrary security groups can be constructed to allow limited, protected data sharing. Protection is enforced per 'buffer plus execution space' pair, because Streamline can check permissions only once, before mapping in the memory region. As a consequence, policy enforcement causes *no* other runtime overhead.

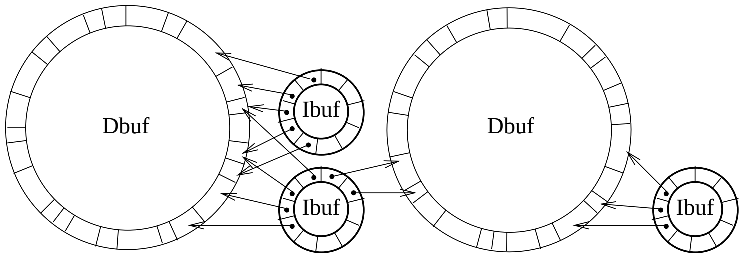


Fig. 5. Chain of data rings holding data and index buffers holding pointers to data

Multiprocessing. Multiprocessing can cause pathological cache behavior, whereby one CPU causes invalidation of lines in another's cache without any real data dependency ("false dependencies"). This is avoided by separating streams and by splitting buffer metadata into private writable and shared read-only memory structures. Similar objectives led to the ring-based netchannel architecture for Linux [Jacobson and Felderman 2006].

Modification. Clients may not modify data stored in *shared* buffers indiscriminately. To avoid data corruption, Streamline detects and resolves read/write conflicts automatically. If it observes a potential conflict, it creates a private writable copy of data and updates the modifying client's pointer (Section 6.1).

For all these reasons, the transport system must support multiple buffers. Interestingly, we can actually exploit this requirement by specializing buffer *implementations* to fit the task profile or hardware at hand. For example, packet reception rings allow overflow, while IPC rings implement blocking semantics. Device driver buffers match the hardware specification of their specific device. In the end, we weave this array of buffers together into a coherent transport system with indirect buffers, to which we now focus our attention.

5.3. Indirection

Presenting clients with an idealized view of private, sequential streams conflicts with copy-avoidance through buffer sharing. A shared packet reception ring holds multiple entangled application streams. When one client needs write access to a shared resource it effectively asks for an independent stream. We can disentangle streams through copying, but that is expensive. The alternative is to use a type of indirection, such as hardware protected virtual memory. Because we do not need isolation (as protection is handled separately by Unix-like access control on buffers), the BMS in Streamline implements indirection in software, to avoid virtual memory management and page faulting. Indirection adds computation, but we will show that the cost is offset by the reduction in data reads and writes. Moreover, the logic is completely handled behind the file interface, i.e., transparent to the end user. We explain how this is achieved in section 5.3.2. First we introduce the core logic.

Streamline replaces pointers with indices and pointer queues with index buffers or "IBufs", that store the indices. Figure 5 shows the interoperation of IBufs with DBufs. IBufs differ from pointers in two ways: they replace direct addressing with globally valid lookup structures and add a small set of metadata fields. Figure 6 shows an example index that references a DBuf, in this case DBuf-2 on NIC1.

5.3.1. Rich pointers. The main feature of indices is their lookup structure: a "rich" pointer. Indices must be able to address buffer contents across virtual memory protection domains. Each index implements a three-level lookup structure consisting of

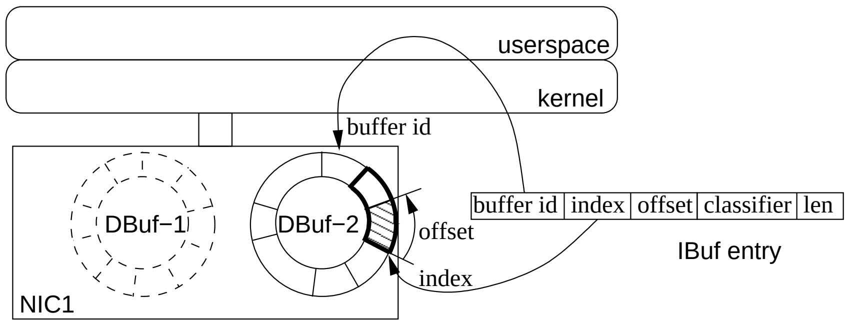


Fig. 6. An index points to a range in a DBuf

a systemwide unique identifier of a DBuf, a block index within this buffer, and an optional offset plus length pair to select a region within the block (e.g., a TCP segment within an Ethernet frame).

Indices from different IBufs may share access to the same DBuf and indices within the same IBuf may point to blocks in multiple DBufs. Figure 5 shows both situations. The first situation is common when multiple clients need a private view on data in a shared ring, which we discussed before. The second situation occurs when a client needs to access multiple rings, e.g., a server listening on two NICs.

Resolving rich pointers is more expensive than following regular pointers, but this cost is amortized by caching a translation for subsequent accesses within the same space. Handling “buffer-faults” is more costly. Such exceptions occur when a referenced DBuf is not mapped into the current memory protection domain. To maintain the illusion of globally shared memory, buffer-faults are handled similar to demand paging: an index pointing to a buffer that is not accessible causes the kernel to map the buffer in the task’s virtual memory (after verifying access permission).

5.3.2. Transparent indirection. The BMS shields clients from indirection details: IBufs present the same file interface as DBufs and perform read- and write-through to referenced DBufs internally, so that clients can remain unaware of which they are accessing. Reading from an IBuf entails resolving the rich pointer and then calling the `peek` method of Section 5.1 of the mentioned DBuf. Writing to an IBuf also involves selecting a DBuf as backing store; currently each space appoints one default buffer. Such transparent indirection enables copy avoidance behind the interface, known as *splicing* [McVoy 1998].

5.3.3. Splicing. When a write request to an IBuf involves data that already resides in a DBuf, write-through can be avoided. This situation occurs often, not in the least because we incorporate the disk cache as a DBuf. On top of IBufs we have implemented splicing: generic, copy-free data transfer between streams.

Splicing has also been integrated into the Linux kernel with version 2.6.17. That implementation differs from ours in two important ways. First, it introduces a new, independent, interface for data transfer and is therefore not backward compatible. Second, it only handles the movement of data within the kernel. While it offers the same performance advantages as Streamline in certain situations, few applications have so far been adapted to the Linux splicing model. Splicing in Streamline, on the other hand, is backward compatible with existing applications. Buffers only present the file interface; applications are unaware of which transfer method is used underneath. During a `write` to an IBuf, Streamline compares the passed source pointer to the address ranges of up to N DBufs, whereby we choose N so that the list of ranges fits in a single cacheline, to minimize runtime overhead. If a list entry matches, Streamline skips write-through, calculates a rich index and only writes this to the IBuf.

Another optimization increases throughput for the common operation of reading a block from one stream and writing it unmodified to another. Splicing only reduces the cost of the `write` call. However, it is also possible to avoid the copy into the application buffer during `read` calls: lazy copying temporarily revokes access to the page(s) underlying the buffer instead of copying. If the application forwards data unmodified, `write` will receive the same application buffer pointer as `read`. The call can then splice from the originating DBuf, instead. Again, Streamline caches the last N pointers for lookup, as well as the accompanying rich pointers. If a page-fault occurs, instead of splicing, data is copied lazily. Splicing is then disabled for the buffer, as frequent page-fault handling actually degrades performance.

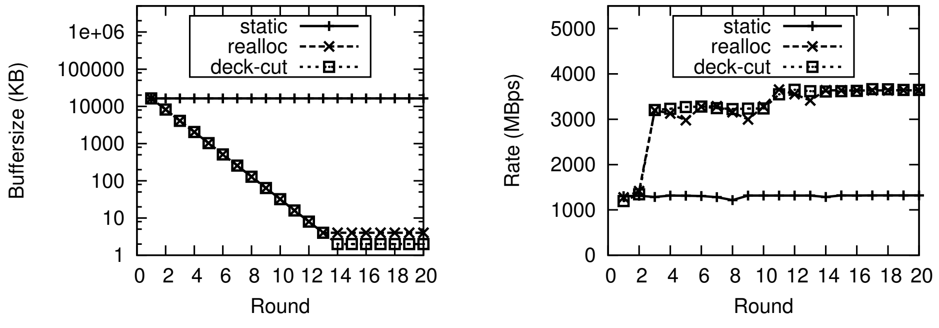
Results. We now quantify the effects of splicing on throughput for both `peek` and `read`, whereby we do not optimize the read call through page access revocation. Figure 4 shows the relative efficiency in transferring data from a DBuf to IBuf, by plotting each method's throughput against call size. The test is indicative of file servers, for instance, where data is read from the page cache and written to the network transmission buffer. The fastest mechanism is *peek only*: the `peek` equivalent of read-only access. This mechanism processes even faster than the physical bus permits, because no data is touched. The method serves no purpose; we only show it to set an upper bound on the performance. About half as fast is *fast peek/write*, which combines `peek` with splicing. This, too, does not actually touch any data, but writes out an IBuf element. Overhead caused by `read` can be seen by comparing these two results with those of *read only* and *fast read/write*. They are 3x slower still. Worst results are obtained when we cannot use splicing, but instead must write out data: throughput drops again, by another factor 2.5. This experiment clearly shows that combined gains from copy avoidance are almost an order of magnitude (9x) when all data is cached. Savings will be even higher for buffers that exceed L2, because then the large blocks will cause many more d-Cache and TLB misses than the small IBuf elements.

5.4. Size

The size of a buffer influences its maximum throughput in two ways: larger buffers reduce synchronization overhead (such as task-switching), but smaller buffers experience fewer cache misses. In ring buffers, miss-rate increases abruptly when a buffer exceeds the size of a cache, because the common (pseudo) LRU replacement policies will consistently evict the last accessed *and thus first needed* slot. On the other hand, task-switch cost (around 1000 cycles on modern CPUs) dwarfs cache miss overhead. Buffers must therefore fit in the smallest cache that does not introduce excessive task-switching. For Ethernet frame-sized blocks, this is usually L2 [Fedorova et al. 2004]. Optimal buffer size can only be determined at runtime, because three factors vary between systems and during runs: memory architectures (number of cache layers, size, etc.), memory system contention, and stream rate and variability.

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

Reallocation replaces one memory region with another of a different size. A reallocation operation can only be started without copying when the producer reaches the end of the region (i.e., when it would otherwise wrap around). As long as consumers are ac-



(a) Buffer size (b) Throughput

Fig. 7. Effects of scaling at runtime, when each round cuts the buffer size in half

cessing the old region, both regions must be kept in memory. The approach is similar to rehashing and has the same drawback: during reallocation the buffer takes up more space than before. Deck-cut avoids this problem. It allocates a maximum-sized buffer, but can temporarily disable parts of it, in a manner similar to how a deck of cards is cut: everything behind the cut is left unused. Deck-cut is computationally cheaper than reallocation, because the only required action is to move the pointer indicating the start of the ring. As a result, it is well-suited to highly variable conditions. We exploit this characteristic by moving the watermarks closer together. A drawback is that it never returns memory to the general allocator.

Scaling is restricted by a few technical considerations. In Streamline, indices (including the read and write pointers) must be monotonically increasing numbers (i.e., they are not reset during a wrap), because those tell in which loop through the buffer – and in the case of reallocation in which memory region – an index falls. To learn the offset of a block in a memory region, one calculates the modulo of the number of slots in the ring (S). When a buffer scales, S changes. To guarantee correct offset calculation for all sizes, modulo operations must always overlap. In other words, all values of S must be natural multiples of the same base. The higher the base, the faster the buffer expands and contracts (we only use base 2).

Results. Figure 7 compares copy (i.e., `write` followed by `read`) throughput for a static buffer of 16MB with that of rings that gradually self-scale down from 16MB until they stabilize. A round denotes a decision moment where the buffer can scale: a moment when the producer wraps around. Figure (a) shows that both scaling buffers continue to decrease buffer size at each opportunity. Figure (b) shows that, instead of scaling linearly with buffer size, throughput sees three levels that correspond with access from main memory, L2 and L1 caches, respectively. The increase in throughput between main memory and L2 is significant: a three-fold improvement. Deck-cut scales further down than reallocation as a result of the moved watermarks.

5.5. Device Interfaces

Many network streams originate or terminate at peripheral devices. Without optimization, a single transfer between devices incurs at least one *horizontal* copy between the peripheral subsystems (e.g., disk and network) and more commonly two *vertical* copies between kernel and userspace, with associated context switches.

Block devices. IO-Lite [Pai et al. 2000] showed that an integrated BMS can remove copy overhead to and from the disk cache. Streamline extends IO-Lite by making these savings available to legacy applications through transparent splicing. Unlike IO-Lite,

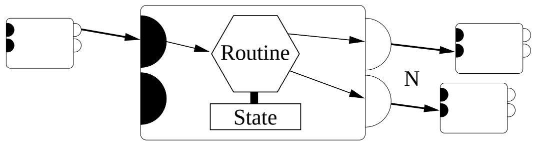


Fig. 8. A filter connected in a network, with one labeled stream

Streamline cannot as yet splice to the cache, because it cooperates with the Linux native disk cache and both systems demand full control over their contents.

Streamline enables read-only splicing from the disk subsystem by exporting cached files as DBufs. Like other DBufs, file contents are locally accessible both from kernel- and userspace. In the kernel, Streamline reflects calls to the Linux virtual filesystem layer. In userspace, it relies on Linux page-fault handling to map in non-cached pages. As a result, it makes a well known performance technique – memory-mapped file access – available to legacy applications.

Network devices. Streamline communicates with network devices through two rings: a reception DBuf and a transmission IBuf. The device driver must implement both, because their formats must agree with all device hardware peculiarities. On the reception path, the protocol stack complements device DBufs with protocol IBufs to demultiplex and reassemble traffic in a copy-free manner. Network devices with hardware classification bypass this logic by directly exporting their own IBufs and protocol filters. Devices with multiple hardware reception rings can export multiple DBufs.

6. PROCESSING

The second main Streamline component concerns processing. Streamline constructs I/O applications on-demand from processing and buffering elements. Such *application tailored-I/O* has the potential to maximize throughput, by moving logic close to data and by exploiting unique hardware features. In the next section we will discuss how Streamline optimizes element placement. First, we present the reconfigurable dataplane that is required to execute arbitrary I/O processing operations in arbitrary configurations efficiently. In this section we present a generic runtime system for executing I/O operations in userspace, kernel and device context and explain why the basic streams and filters model must be extended to support the various classes of operations efficiently, especially across multiple software tasks and hardware devices.

6.1. Extended Streams and Filters

A common model for expressing reconfigurable streaming I/O is that of *streams and filters* (e.g., [Ritchie 1984; Hutchinson and Peterson 1991; Montz et al. 1994; Welsh et al. 2001; Kohler et al. 2000]), which defines a directed graph where vertices represent I/O operations, or *filters*, and edges symbolize I/O transport means, or *streams*. Streamline implements this model and extends it with the concept of execution spaces, to be able to automate the mapping of software onto hardware. A pure SF graph only describes an I/O path *request*. An I/O path *implementation* additionally binds each filter to an execution space.

Streams. Streams are point-to-point channels between filters. They map trivially onto ring buffers, to form a store and forward network where each data block is saved at each edge. But, the reading and writing of indices – let alone data blocks – exceeds the operational cost of many I/O operations. Streamline avoids this unnecessary storage cost in the common case. If two connected filters execute in the same memory protection domain and no external parties (such as controlling application logic)

Table I. Filter classes and their requirements

Class	**Behavior**	**Requirement**
Source	Bootstraps graph walk	Callback interface
Pass-through	Stores metadata	Random-access storage
Inspection	Analyzes stream contents	Read-only (shared) stream access
Filtering	Classifies stream contents	Class-tagged edges
Reordering	Modifies stream indirectly	Indirect buffers
Rewriting	Modifies stream contents	Read-write (private) stream access

require on-demand access to the interconnecting stream, it schedules the second filter immediately after the first and passes a pointer in-memory rather than saving an index (or the original payload) to an intermediate buffer.

Merging filter execution at runtime in this manner increases data cache hitrate and removes non-functional scheduling and memory access overhead. The optimization can be applied recursively throughout a request graph. In practice, Streamline optimizes away nearly all ring buffer accesses, because memory protection crossings are few and application logic is commonly interested only in a single stream of the I/O path: the end-result of all transformations.

Filters. Filters are self-contained routines that can only communicate with each other through a set of input and output ports to which streams are attached. Incoming data may arrive on any input port; outgoing data is sent on all output ports. In contrast to computational kernels, filters may access private state. Figure 8 depicts a simple configuration. The filter logic is unaware of the number of connected inputs or outputs. The filter is instantiated by registering it as the callback to a stream. Each filter can attach to multiple streams, for which it exports a set of ports. When a block arrives on an input port, the runtime system calls the filter's routine.

Streamline carries 40-odd filters, implementing such diverse tasks as protocol processing (fragmentation, multiplexing, checksumming), filtering (pattern recognition, anomaly detection), transformation (compression, encryption), accounting, logging and tunneling. We categorize these into six broad structural classes according to their behavior. We briefly discuss each class, present an example filter and indicate what requirements it places on the datapath beyond store and forward. Table I summarizes the results.

Sources inject blocks into the graph. In practice, they are usually callback functions on external I/O hardware, such as the soft interrupt handler of the Intel `e1000` device driver. Streamline exports both a simple DBuf injection routine for legacy drivers and the more efficient zero-copy driver API described in Section 5.5 that fully integrates device memory as DBufs and descriptor rings as IBufs.

Pass-through filters only process metadata found in indices. This class includes counters and histograms (e.g., of inter-arrival times). These filters require private memory that persists across invocations to store their metadata and a mechanism to communicate this information to the end-user. Streamline reuses memory regions for this task, as these already implement both requirements.

Inspection filters derive metadata by analyzing stream contents. For instance, `ipfix` builds a connection database from observed IP packets. These filters require shared read-only access to streams, which the BMS provides.

Filtering is based on inspection of streams to extract values and combines it with selection. Applications label edges between filters with class ranges and data is only forwarded if its value falls within the range. The `ipproto` filter splits traffic according to TCP/IP protocol number.

Reordering is the modification of streams through indices alone. The quintessential example is TCP reassembly. Besides a classical implementation that moves data, Streamline also carries a *zero-copy* reassembly filter, which modifies indices' offset and length fields and in case of out-of-order arrival reorders indices instead of payload.

Rewriting occurs when data transformation cannot be captured in indices. Filters in this class must have read/write access to their streams. In a shared-memory system write access must be restricted, however, to guarantee data consistency. Streamline requires consumers to call a gateway function before modifying a block. The function uses runtime system information to prepare a private reference in the most efficient manner (direct access, copy-on-write, immediate copy), a choice that depends on concurrent clients and block size. If a block becomes fragmented as a result of selective copying, the runtime system automatically calls the downstream vertices for each fragment.

Spaces. Automating the task of selecting an execution environment for a filter requires a model of the local computer architecture. For this purpose, we extend the streams and filters model with the concept of *execution spaces*: combinations of an execution thread and a memory protection domain. On the x86 architecture, a space maps one-to-one onto a single threaded user process. On peripheral devices, memory and processing capabilities may be more limited, for instance lacking virtual memory or multitasking. Concrete examples of execution spaces include Unix processes, kernel tasks, GPU stream processors and a multi-ring NIC's integrated hardware switch.

6.2. Execution

The set of active execution spaces across user processes, kernel tasks and device contexts forms the runtime system of Streamline. Each space has a local scheduler queue, on which it schedules filters as data arrives from sources or as it filters traffic on streams. This model is particular instance of user-level threading. The express goal is to replace the task switching common in Unix pipelines by function calling. Spaces have CPU affinity to be able to reason about global cache use. While BMS design impacts copy overhead, the runtime system strongly influences context switching cost: because the system schedules downstream filters, it can minimize this overhead by batching signals – in as far as buffer size permits. The runtime system also decides whether to use fast local function calling between filters, or whether to save the index to an IBuf and call downstream filters by sending an explicit signal. Across execution spaces, the second, slower method is required.

Intra-space. The subset of the I/O path that executes within a single execution space is implemented as a single event loop. Itself started by a source, this loop recursively schedules downstream filters in breadth-first order as long as the filters' classification results match their downstream edge labels. By calling filters in rapid succession for the same datablock, data cache hitrate is maximized at the cost of the instruction cache. For most graphs, this heuristic indeed gives the most efficient solution, because the combined instructions tend to fit in the instruction cache together. StreamIt [Sermulins et al. 2005] implements a more refined scheduling policy.

Inter-space. Across execution spaces function calling is not an option. Here we must resort to another, more expensive, method. Polling and interrupt driven processing are standard approaches, but both have drawbacks: polling wastes cycles at low rates and interferes with scheduling; interrupts incur cost at high rates, exactly when the system is already stressed. Hybrid systems, such as interrupt moderation, clocked interrupts [Traw and Smith 1993], or interrupt masking (as implemented in Linux NAPI [NAPI]), evade both pathological cases. Streamline combines interrupt moderation with timeouts to amortize cost at high rates while bounding worst case delivery latency. Its approach is unique in that both the interrupt moderation threshold and timeout value can be set individually for each buffer. This way, the trade-off between efficiency and latency can be tuned to application constraints.

Results. Figure 9 shows pipe throughput offset against call size at various levels of interrupt moderation factor. All results are obtained with a large DBuf of 16MB, or 8000 slots, so that we have a wide measurement range for signal moderation. Streamline sends at least one signal per timeout epoch (in the experiment set at 1000HZ) which limits moderation benefit for very high numbers. The figure shows that, indeed, throughput scales below linear. When batching up to 32 signals we already achieve 92% of the maximally obtainable gain (with factor 2048): 2.36x versus 2.56x. For this reason, 32 is the default moderation factor in Streamline. For IBufs tuning is more involved because we must also prevent overflow of referenced DBufs. A simple and safe heuristic is to set the threshold to that of the smallest referenced DBuf.

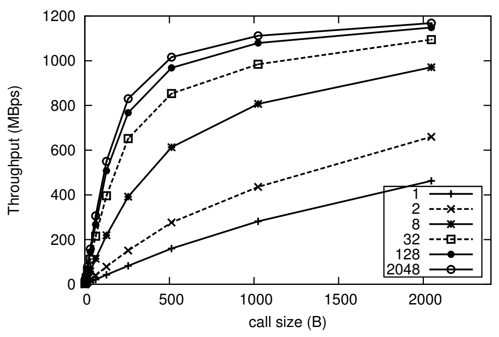


Fig. 9. Interrupt moderation

7. CONTROL

Performance of a pipeline is largely determined by how the I/O path is mapped onto the local computer architecture: decisions to use function calling over buffering or to move a function from a process to a device can produce an order of magnitude increase in throughput for a local segment of an I/O path. But, application of scarce device hardware to the wrong filter can depress end-to-end throughput. Streamline automates the mapping operation from pipeline onto I/O path to consistently make good (if not necessarily optimal) decisions. This section completes the description of the three main components in Streamline (buffering, processing, and control) with the discussion of the automatic control system.

Figure 10 shows the control architecture in Streamline. At the application level, a parser breaks up the pipeline expressions into a sequence of operations on individual filters and streams, for example finding all implementations of a filter (discovery) or attaching a stream to an IBuf (allocation). Spaces locally account their available resources and manage their own scheduling and forwarding. They respond to discovery and allocation requests over a control RPC mechanism. A simple packet-switched messaging network interconnects spaces to relay messages from applications down to kernel tasks and devices.

7.1. Algorithm

The I/O path construction process consists of three phases: discovery, selection and allocation. Each phase must successfully complete for all filters in the pipeline before the next phase is entered.

Discovery. The first phase discovers all potential filter implementations. A regular expression filter, for instance, is implemented in each userspace task, but some high-performance NICs can cheaply test multiple patterns using content addressable memory or FPGAs [Hruby et al. 2007]. The discovery phase returns a graph with multiple candidate filter implementations for each filter.

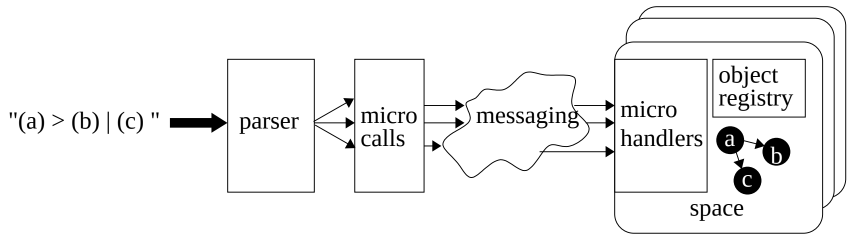


Fig. 10. Control architecture

Selection. Filters with multiple implementations are the basis for adaptation in Streamline. A selection algorithm prunes a discovery graph that holds sets of candidate implementations for nodes into one with a single implementation per node. In this paper, we eschew complex optimization in favor of a simple heuristic: *push processing down*. Spaces are connected more-or-less vertically, from software-isolated applications at the top to dedicated peripherals such as NICs and GPUs at the bottom. Lower placed spaces generally have access to specialized logic (e.g., ASICs) that makes them both tailored to a matching task and useless to all others (in contrast to the contented CPU time higher up). As a result, the algorithm chooses the "deepest" implementation. For simplicity, the algorithm is greedy: the best match is selected for each filter, irrespective of global concerns. The only constraint is that I/O paths may only grow from their source(s) in the direction of their sink(s). All graphs originate at a few natural sources (network cards, switches, applications, disk) and flow to equally well-defined sinks. By forcing a single direction for all intermediate I/O paths, we ensure that no loops or ping-pong effects will occur and therefore transport cost is minimized. Push processing down is easy to understand and performs well in networking scenarios. However, for robust execution in wider domains, we also investigated more high fidelity models for which linear solvers compute globally optimal solutions in tens of milliseconds [de Bruijn 2010].

Allocation. When a collection of filters and streams has been found to satisfy a request, the elements are allocated and configured by a succession of small construction requests. Chance of partial failure is high, because allocation takes places across multiple spaces that are partially independent. For one, concurrent application requests are not serialized. End-to-end pipeline allocations are transactional, to be able to retract in case of unexpected resource conflict. To achieve this, Streamline requires each individual allocation call to be atomic, have only binary outcome and have an inverse operation (if not idempotent). All steps are logged and, in case of failure preceding steps, are unwound. Allocation steps fall into three phases. First, filter implementations are instantiated into runnable filters, which combine logic with private state. Next, streams are attached to the ports. If a stream crosses spaces, output buffering is enabled and a signaling path is set up (with moderation). Finally, the filters are activated so that data can start flowing. Activation occurs back to front, from the I/O path sinks to its sources, so that the entire new path is enabled at once without a need for a global lock. Streamline applies prefix sharing between I/O paths to further reduce runtime cost. As pipelines are added to the runtime system, segments that overlap with existing I/O paths are mapped onto those. Filters benefit from sharing vertices when they need to perform identical initial steps. This is a common occurrence, for instance, in protocol stacks, where each socket receive path will want to observe all packets and filter out its own. Overlapping pipelines has the effect of implementing only a single protocol demultiplexer. To avoid inconsistent state, only stateless or newly initialized filters may be shared.

8. SECURITY

Streamline gives the same level of protection as Unix, but can also trade isolation for throughput on a case-by-case basis. On many modern computers systems, more data can be safely exported to userspace than is current practice. One example is in network receive processing. Raw packets arriving from the network are by default not shared on most operating systems to maintain privacy. On a multi-user system where users cannot access the physical medium the OS indeed is a data access choke-point. More and more, reception queues can be safely mapped, however. This is certainly true when (1) all data in a reception buffer is destined for the same application or (2) isolation is already pierced elsewhere. The first case can hold on systems with multiple or multiring NICs. Even without such hardware, it is often the case: on dedicated networked servers that perform a single task, only administrator data is mixed in with application data. If all superuser traffic is encrypted, as for instance SSH sessions are, all network packets can be safely mapped read-only into the application context. The second case occurs when the network outside the host cannot be trusted, in other words: for all Internet connections.

In this common case, reception queue isolation through copying incurs cost without offering real protection. Instead, rings can be mapped in one of two ways. Read-write mapping enables in place modification by applications, but cannot offer integrity protection when multiple distrusting processes share a buffer. Read-only mapping saves the copy operation and offers integrity – if not confidentiality. As few applications perform in-place modification of receive traffic and the physical network generally also punctures privacy, this second option is the default.

Receive queue sharing is one example of the general case where composite systems such as Streamline enable *selective* access to data and computational resources (filters). Streamline implements access control by applying Unix permissions to kernel and device objects. To this goal, Streamline treats spaces as (active) access control subjects and filters, streams and shared buffers as (passive) objects. In line with Unix practice, spaces have an identity made up of one unique user account and an optional set of groups to which the space belongs, denoted by a UID and set of GIDs. Space UID and GID derive from their controlling user for userspace processes. Kernel spaces are owned by the root superuser. A filter is owned by its space, but the stream it produces is owned by the UID that requested instantiation of the filter. With fan-in, stream permissions correspond to the union of all inputs.

Streams. The use of shared buffers introduces security perimeters independent of virtual memory protection domains. As we explained in the communication section, buffers are mapped into a domain once for the duration of use. Protection is enforced per 'buffer plus computation space' pair. Streamline checks permissions only once per pair: just before mapping a buffer into a space, which can occur during a call to `open` or as result of a buffer-fault (similar to a page-fault). Because this is the only occasion when computation spaces are given access to a new security perimeter, it is a security 'choke point' and thus acceptable as the sole policy enforcement gate.

To minimize data copying and virtual memory operations we aim to share buffers between streams. Multiple streams can safely be stored in the same buffer as follows: first, streams with identical ownership and permissions can safely coexist in the same buffer. Second, streams that belong to the same group and that grant the same permissions to their group as to their owner can be co-located. Finally, streams that are accessible by all can share a buffer.

Filters. The model of application paths that extend into the kernel collides with the practice of isolating users. The system has to defend both the integrity of the kernel and the privacy of user data. In monolithic network stacks only the superuser may load extensions into the kernel or onto device hardware. In Streamline, the root may relax permissions on BPF or other trusted kernel filters to be executable by other users. Applications written in languages such as BPF, even if inserted by malicious users, cannot corrupt the system. In this case, security derives from the use of safe filters, including programmable safe language interpreters and compilers. When a filter has multiple implementations in different spaces, an application that lacks privileges to access the efficient implementation on a peripheral device will perhaps witness a drop in performance when having to an the application-level version, but not necessarily a drop a functionality. For example, if a hardware MPEG decoder is protected, the pipeline falls through to a software implementation. The `tcpdump` analyzer is protected on Unix, but Streamline safely gives users the tools to inspect their own communication streams, possibly with BPF and `tcpdump`.

9. EVALUATION

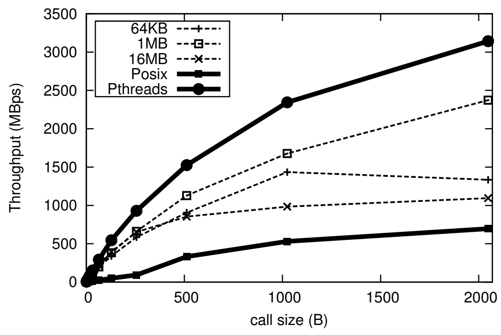
We compare a Streamline-enabled version of Linux 2.6.24.2 head-to-head with a stock version in terms of application throughput (or CPU utilization at steady rate). All tests were run on an HP 6710b with Intel Core 2 Duo T7300 processor, 4MB L2 cache and 2 GB RAM running in 32-bit mode. We ran the experiments on a single core to minimize scheduler influence and show the full cost of task switching.

In the previous sections we supported our claims by micro-benchmarks where applicable. The presented numbers substantiate the third of the four quantitative contributions claimed in the introduction. We now present end-to-end results that correspond to the other three claims, in order. The first, Unix primitives, displays the raw gains at the buffer interface level. The second, application benchmarks, shows achievable gains for applications that use legacy interfaces. The last, native applications, indicates how the native interface expands application freedom and performance.

9.1. Unix primitives

Figure 11 shows throughput of straightforward copying (a `write` followed by a `read`) through a Unix pipe at varying buffer size and for three IPC implementations: a standard Linux pipe (with label ‘posix’), a producer/consumer pair of threads that directly access shared memory (‘Pthreads’) and streamline buffers of varying size. In this test, we do not use Streamline’s `peek` optimization and thus copy the same amount of data as the other applications. Any performance improvement comes from a reduction in context switching. The threaded application shows an upper bound on achievable performance, because it requires no kernel mode switches at all and it implements a multi-packet ring. Similar to Streamline rings, its throughput is dependent on buffer size, but we only show the best case here for clarity (1MB). That configuration outperforms Linux’s implementation of Unix pipes by a factor 5 for large blocks and 12 for minimal blocks. In between are 4 differently sized Streamline tail-drop DBufs. We see that the fastest implementation is neither the largest (64MB), nor the smallest (64KB), but an intermediate one (1MB). This outperforms Linux by a factor 4 for large and 9 for small packets and is only between 20 and 33% slower than the optimal case. The precise factor of throughput increase depends on physical cache size, producer-consumer distance and whether the application buffer is cached, but the ratios are static; we previously observed similar results on different hardware [de Bruijn and Bos 2008a].

Figure 12 explains why the highest throughput is achieved with a medium-sized buffer. Initially, performance grows with the buffer as the number of necessary context switches drops when calls are less likely to block. Ultimately, however, page-faults affect performance as the datacache or TLB begins to witness capacity misses. These



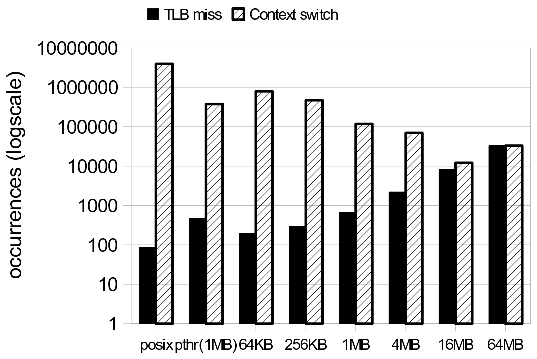


Fig. 11. Unix pipe throughput: rate of various Streamline configurations compared to threads (best case) and processes (worst case).

Fig. 12. Pipe cost factors: number of task switches and cache misses observed at various buffer sizes.

are more expensive than switches, therefore maximum throughput is obtained when the working-set just fits in the L2 cache.

9.2. Application benchmarks

The micro-benchmarks demonstrate that significant savings in I/O overhead can be achieved by optimizing buffer parameters and employing copy avoidance. We now investigate to what extent these savings translate to real application domains. For this purpose we ran the applications introduced in Section 2 on top of both the Linux and Streamline versions of sockets and libpcap: `bind` identifies per-call overhead, `mplayer` per-byte overhead, `tcpdump` parallelization cost and `dd` splicing effects.

9.2.1. DNS serving with **bind**. The Bind `named` daemon replies to DNS requests. We ran a version 9.4.1 daemon in non-recursive mode and sent it a steady stream of 10 thousand requests per second. DNS messages are small: requests were below 55B and replies below 130, including IP headers. For this reason, application processing easily dominates total overhead. Figure 13 shows total CPU utilization for Linux and Streamline, whereby for Streamline we vary both buffer size and moderation threshold (we plot Linux for multiple values for clarity only, it does not throttle). We present the median of 103 measurements, with upper and lower quartiles within 20% of the presented results.

The figure shows that the daemon performs between 1.1x and 1.5x as well on top of Streamline as on top of Linux. As expected, this factor grows with the amount of moderation. Buffer size affects performance less. This is a result of hardware constraints of the specific Intel Pro/1000 NIC that restrict buffer scalability freedom: the reception DBuf must be at least 1MB and the transmission queue at most 1MB.

9.2.2. Video reception with **mplayer***.* To demonstrate savings for applications that handle large blocks we now present results obtained with streaming a high definition (SXGA), high data rate (100Mbit) MPEG4 stream to a popular video client: mplayer 1.0 RC2. We disabled software decoding of the compressed stream, as the (software) decoder would render the application computation-bound. Our results thus compare to machines with hardware video decoders. Figure 14 summarizes the results obtained with 1MB reception and transmission buffers (again, chosen because of NIC requirements). Here, too, Streamline is more efficient than Linux, between 2- and 3-fold, depending on moderation threshold.

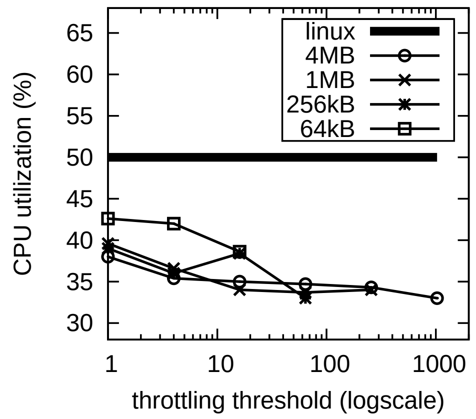


Fig. 13. Bind: CPU utilization at 10kreq/s

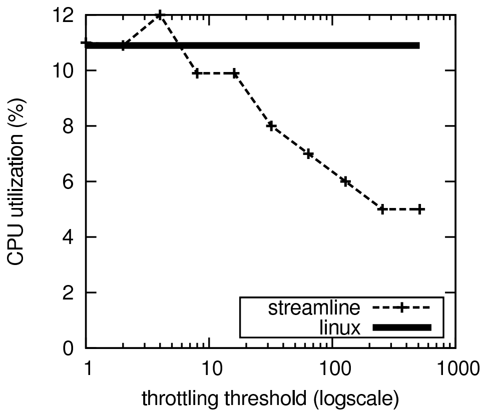


Fig. 14. Mplayer: cpu load for 100Mbps sxga

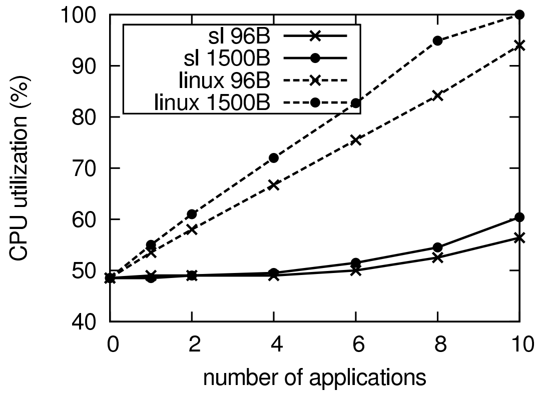


Fig. 15. Tcpdump: cpu load for 200Mbps flow

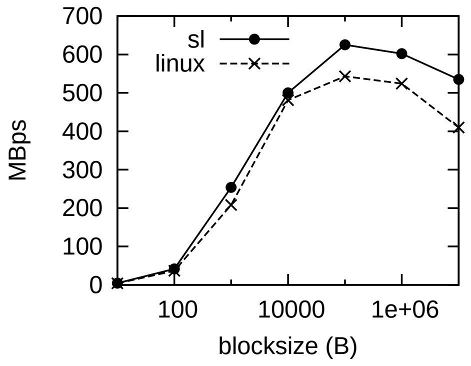


Fig. 16. dd: throughput at varying blocksize

9.2.3. Network capture with **tcpdump**. Figure 15 shows throughput of tcpdump 3.9.8, a popular traffic analyzer. To investigate scalability with parallel data access, we capture a moderate datastream: 200 Mbit of full-sized 1500B packets per second, generated with `iperf 2.0.2`. The iperf server requires 50% CPU time. When capturing with a single listener, Streamline ('sl 96B') uses up hardly any extra resources, while standard Linux ('linux 96B') requires 5% CPU time (10% of the application cost). Savings decrease as we run applications in parallel. When capturing full frames with 10 instances, Streamline causes a 13% CPU overhead, slightly above a single Linux instance, whereas Linux saturates the CPU and drops packets. Running ten instances of tcpdump is not a common operation, but the results are representative for any system configuration where multiple applications access the same data, for instance network intrusion detection and group communication using multiway pipes.

9.2.4. Disk Transfer with **dd**. A common server application pattern is to transfer data from the disk (cache) to the network stack. Splicing traffic from the cache will theoretically increase throughput by avoiding up to two copies: on read from disk into the application and on write to the network stack. We isolate the effects of splicing by comparing throughput of a pure disk application. Figure 16 plots throughput of the popular Unix `dd` tool while copying a 50MB file at increasing blocksizes (the '`bs`' argument to the tool). Both cases write data to a regular file, therefore only one copy is saved. The figure compares copying from the Linux pagecache to splicing from a Streamline DBuf. Throughput gains lie between 3.8 and 23%, with 14% at the optimal blocksize. Upper and lower quartile lie within 6% of the median of 11 runs.

9.2.5. Limits: latency with **tftp+**. We observed the limits of Streamline optimizations when porting A UDP-based fileserver. **tftp+** sends multiple data packets per acknowledgment, at a fixed ratio defined by the client. Batching is ineffective because control is latency bound. With small tftp blocksize (512B), so is splicing. Both systems observed a CPU load of approximately 5% when handling a 64 Mbps (16000 pps) connection.

9.3. Native applications

Applications programmed against the native pipelines observe the highest throughput and maximum portability, because only pipelines are fully reconfigured. We have built four network applications directly on top of Streamline to demonstrate non-standard I/O paths. Because these are complex applications that have no comparable legacy counterparts, we omit full quantitative results and refer to the corresponding application-specific publications.

SafeCard [de Bruijn et al. 2006] is a network intrusion prevention system for edge hosts that combines full packet payload scanning, application-layer protocol filtering (which requires traversing the entire protocol stack) and flow-based behavioral detection. It is implemented as a single Streamline I/O path that can run in software, but (to protect at full Gigabit speed with minimal overhead) also runs as a combined software/hardware path on an Intel IXP2400 smart NIC. There, zero-copy TCP reassembly and pattern matching filters are offloaded to specialized stream processors. Application-layer and behavioral detection execute as software filters, because they are complex to write and not computationally demanding.

The **token-based switch** [Cristea et al. 2007] implements traffic prioritization based on cryptographic tokens embedded in each packet. To perform the necessary cryptographic operations at multi-Gigabit line rate, we built the switch using Streamline and ported it to run on a dual Intel IXP2850 network processor, which besides a CPU embeds 32 specialized stream processors and two hardware cryptography units. The datapath, including token insertion and verification modules, is passed as a single I/O path. Streamline maps the filters onto the stream processors and crypto-units and interconnects all through shared ring-buffers.

Webtap is an application-layer traffic monitor. It attaches to the network reception path, scans for HTTP requests and writes summaries to a logfile, all within the kernel. Logging is often disabled on production webservers, because it introduces a non-negligible performance hit. Webtap was developed in cooperation with Wikipedia to run in their datacenters (but could not be installed due to confidentiality concerns), reconstructing requests read-only from a switch's mirror port. By using zero-copy TCP reassembly and executing completely in the kernel, webtap implements a minimal I/O path that in vitro showed to scale to the 10K requests per second [wik 2008].

Nemulator is a sophisticated detector of code injection attacks in the network based on Nemu [Polychronakis et al. 2007]. Rather than looking for specific patterns, Nemu detects an attacker's code (the 'shell code') by treating every byte in the payload as its potential entry point. Thus, it will execute these bytes as if they were instructions. Typically, the execution hits an illegal instruction fairly quickly and Nemu will try again, with the next byte and so on. It raises an alert whenever the execution behaves in way that is indicative of shell code (such as running getPC sequences and executing bytes that it just wrote). Clearly, executing every byte in the payload is very expensive, and performance has been limited to a few tens of Mbps when checking the full

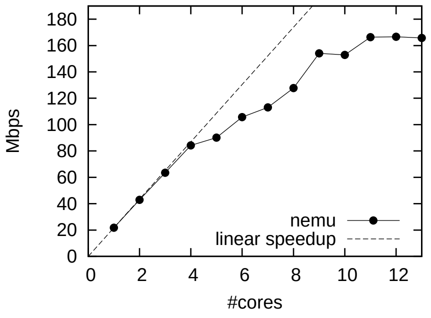


Fig. 17. Nemu scalability with cores

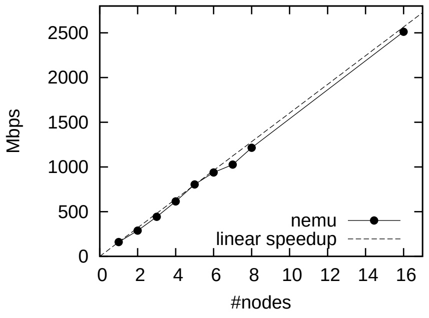


Fig. 18. Nemu scalability with machines

stream in this way. With Streamline, we built a parallelized Nemu implementation for multicore systems. A streamline pipeline reassembles ingress TCP traffic and divides the connections among a fixed set of streams. Each stream is communicated by shared memory to a single core running a Nemu process. On a single Xeon X5650, 2.67GHz, 6-core machine with two threads per core, this configuration achieved 15-20 Mbps per thread and an overall throughput of 170Mbps with 11 processing threads. Figure 17 plots throughput with increasing numbers of cores. When spreading the task over multiple machines (replacing shared memory with UDP tunnels), the system scales to Gigabit rates. Figure 18 shows that this slightly different configuration observes linear scalability and reaches 2.5Gbps of aggregate throughput at 16 nodes.

10. RELATED WORK

Streams [Ritchie 1984] is the canonical streams and filter network stack. Click [Kohler et al. 2000] applies S&F to routing. It operates at layer 3, where no per-session or per-flow state has to be kept and paths are therefore static. SEDA [Welsh et al. 2001] is an application layer network server architecture that reconfigures the network to scale to distributed hardware. Dryad [Isard et al. 2007] also maps a directed graph of operations to distributed resources. It automates resource selection, but unlike Streamline uses this for scalability and fault-tolerance in cluster environments. The x-kernel [Hutchinson and Peterson 1991] is an OS network stack that extends from application to devices and implements fast I/O channels between these contexts. Scout [Montz et al. 1994] extends x-kernel with QoS support for real time applications and is available as a Linux kernel module [Bavier et al. 2002]. *Only the x-kernel and Streamline represent end-to-end operating system network stacks. Streamline differentiates itself in from all systems in four ways: it (1) applies the Unix programming model to network processing and thereby (2) presents a single elegant model for application, kernel and peripheral programming, (3) minimizes memory access cost through a novel static ring BMS and (4) automates the optimization to variation in hardware. This is the first work to propose a layered I/O architecture with independent buffering, processing and control systems.*

Unix Network Programming

Streamline applies the Unix philosophy that "everything is a file" to the networking domain and the kernel and peripheral contexts. Network connections in Plan9 are also based on filesystem nodes, but use special operations (e.g., `dial`) and do not expose intermediate streams in the network stack in the manner of PipesFS [Presotto and Winterbottom 1993]. *Streamline is the first to demonstrate a high-throughput implementation of Unix primitives for network programming. Specific architectural contri-*

butions are the PipesFS virtual filesystem for streaming I/O, the extended shell syntax, a pure filesystem basis for sockets and Unix permissions for safe kernel programming. Streamline is the first system that safely grants zerocopy application access to selected network queues, bar pure capture cards.

Peripheral Programming

Scout [Montz et al. 1994] has been used in conjunction with a programmable NIC to build an extensible layered router, Vera [Karlin and Peterson 2002]. Like Click [Kohler et al. 2000], Vera works at the network layer. It uses layering similar to execution spaces, but in Vera I/O paths are handcrafted. Spin [Bershad et al. 1995] moves application logic to the kernel through the use of a safe language, extensible interfaces and resource management. Spine [Fiuczynski et al. 1998] further applies these concepts to programmable network cards. The solution is similar to Streamline's filter offloading to peripheral hardware [Nguyen et al. 2004], but restricts itself to single safe extensions and NICs. *Unlike these systems, Streamline extends the same programming model to the peripheral. It imposes no restrictions on number of logical blocks or streams beyond those imposed by hardware. Specific contributions include the messaging network that spans across 'execution spaces' as a control plane and the static buffer design that seamlessly integrate hardware descriptor rings.*

Copy and Context Switch Avoidance

Streamline minimizes three causes of non-functional I/O overhead: copying, context switching and cache misses. It reuses well known I/O optimizations, such as efficient upcalling [Clark 1985] and inter layer processing [Clark and Tennenhouse 1990]. Copy-avoidance mechanisms generally replace copying with virtual memory (VM) techniques such as page sharing and copy-on-write; Most such techniques work at the granularity of single blocks. While cheaper than copying, recurrent modifications to VM structures still contribute cost to each data transfer. Brustoloni [Brustoloni and Steenkiste 1996] categorized previous efforts and showed them to perform roughly identical. In the context of x-kernel, Druschel *et al.* describe copy avoidance ideas for network buffers [Druschel et al. 1993] and subsequently translate these into Fbufs [Druschel and Peterson 1993]: copy-free communication paths across protection domains that remove the per-block costs in certain situations. Paths are efficient only if mappings can be reused between blocks. Fbufs were later incorporated into IO-Lite [Pai et al. 2000], a BMS that replaces copying by updates to a mutable structure of pointers to immutable buffers. The Streamline BMS, on the other hand, completely avoids per-block operations by extending shared ring buffers [Govindan and Anderson 1991] with specialization (among which indirection), compression and backward compatible access control. Xen exchanges descriptors between domains through rings to reduce switching, but does not allow shared data rings [Fraser et al. 2004]. Tribeca [Sullivan and Heybey 1998] is a stream database (a query system for continuous data streams) that also employs indirection, but operates in a different domain (e.g., financial records) and faces different bottlenecks (disk I/O). *Streamline is the first general purpose OS I/O architecture that incurs* no *copy or vmm operations at runtime at all. It demonstrates that a general purpose design based on long-lived static rings is not just feasible, but fast. Specific contributions are the introduction of statically mapped pointer rings (IBufs) for selective zerocopy communication across protection domains and an implementation of splicing that is backward compatible with Unix I/O interfaces and therefore transparent to (legacy) applications.*

Automatic Configuration

Self-optimization of operating system logic is not a new idea [Blevins and Ramamoorthy 1976], but application is rare in practice. Related work comprises three categories: implementation specialization, communication optimization and code relocation. The first uses some for of reflection to select an operation implementation at runtime. Synthetix [Pu et al. 1995] specializes I/O system call implementations (e.g., `read`), similar to Streamline buffers. Unlike Streamline, it also automatically chooses the best matching implementation based on application invariants. Object-oriented (OO) operating systems such as Apertos [Lea et al. 1995] and Choices [Campbell et al. 1991] use late-binding of objects, but have no selection criterion to choose among implementations and operate on a single object (as opposed to a composite I/O path). Amoeba [Tanenbaum et al. 1990] is a distributed OO OS that selects optimal communication channels based on object location. Flex [Carter et al. 1993] does the same, but also moves logic between address spaces if this reduces communication overhead. Lipto [Druschel et al. 1992] and Kea [Veitch 1998] also allow relocation of logic between address spaces to reduce communication overhead. EPOS [Fröhlich and Schröder-Preikschat 1999] and Knit [Reid et al. 2000] construct operating systems from components. All presented systems are reconfigurable, but none automate the optimization step. Ensemble [Liu et al. 1999] automatically prunes a network stack to create a faster "bypass" version by identifying context-dependent simplifications. *Only Streamline fully automates the translation from application specification to running implementation. It demonstrates a method, "push processing down", that is simple to reason about and efficient enough for online use. Additionally, the control layer built from simple messaging and fault tolerant provisioning serves as an early implementation of the distributed system model of OS design [Schapbach et al. 2008].*

11. CONCLUSION

By tailoring I/O paths automatically and on demand, Streamline takes an extreme position in the OS design space. The architecture reduces overhead from copying, context switching and caching, which improves throughput of a representative set of applications between 30% and 10x over standard Linux. Furthermore, it presents an OS-based solution to the problem of integrating special-purpose hardware.

Streamline demonstrates that extensibility does not automatically force complexity on the end-user or application developer. Through a concise language and heuristic-based optimization it relieves them of all technical detail. The result is practical software that can be, and has been, directly applied to networking tasks such as intrusion prevention, application serving and media streaming.

Acknowledgments

We are grateful to Kees Verstoep and the anonymous TOCS reviewers for commenting on earlier drafts of the paper, Hamid Bazoobandi for helping with the evaluation of splicing, and Tudor Zaharia for the Nemulator application.

REFERENCES

2008. Wikipedia statistics. http://en.wikipedia.org/wiki/Wikipedia:Statistics.

BAVIER, A., VOIGT, T., WAWRZONIAK, M., PETERSON, L., AND GUNNINGBERG, P. 2002. Silk: Scout paths in the linux kernel. Tech. rep., Uppsala University.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. 1995. Extensibility, safety and performance in the spin operating system. In *15th Symposium on Operating Systems Principles*. Copper Mountain, Colorado, 267–284.

BLEVINS, P. R. AND RAMAMOORTHY, C. V. 1976. Aspects of a dynamically adaptive operating system. *IEEE Trans. Comput. 25,* 7, 713–725.

BOS, H., DE BRUIJN, W., CRISTEA, M., NGUYEN, T., AND PORTOKALIDIS, G. 2004. Ffpf: Fairly fast packet filters. In *Proceedings of OSDI'04*.

BRUSTOLONI, J. C. AND STEENKISTE, P. 1996. Effects of buffering semantics on i/o performance. In *Operating Systems Design and Implementation*. 277–291.

CAMPBELL, R. H., ISLAM, N., JOHNSON, R., KOUGIOURIS, P., AND MADANY, P. 1991. Choices, frameworks and refinement. In *1991 International Workshop on Object Orientation in Operating Systems*. 9–15.

CARTER, J. B., FORD, B., HIBLER, M., KURAMKOTE, R., LAW, J., LEPREAU, J., ORR, D. B., STOLLER, L., AND SWANSON, M. 1993. FLEX: A tool for building efficient and flexible systems. In *Proc. 4th IEEE Workshop on Workstation Operating Systems*.

CLARK, D. D. 1985. The structuring of systems using upcalls. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 171–180.

CLARK, D. D. AND TENNENHOUSE, D. L. 1990. Architectural considerations for a new generation of protocols. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*.

CLEARY, J., DONNELLY, S., GRAHAM, I., MCGREGOR, A., AND PEARSON, M. 2000. Design principles for accurate passive measurement. In *Proceedings of PAM*. Hamilton, New Zealand.

CRISTEA, M.-L., GOMMANS, L., XU, L., AND BOS, H. 2007. The token based switch: Per-packet access authorisation to optical shortcuts. In *Networking*. Lecture Notes in Computer Science Series, vol. 4479.

DE BRUIJN, W. 2010. Adaptive operating system design for high throughput i/o. Ph.D. thesis, Vrije Universiteit Amsterdam.

DE BRUIJN, W. AND BOS, H. 2008a. Beltway buffers: Avoiding the os traffic jam. In *INFOCOM 2008*.

DE BRUIJN, W. AND BOS, H. 2008b. Pipesfs: Fast linux i/o in the unix tradition. *ACM SigOps Operating Systems Review 42,* 5. Special Issue on R&D in the Linux Kernel.

DE BRUIJN, W., SLOWINSKA, A., VAN REEUWIJK, K., HRUBY, T., XU, L., AND BOS, H. 2006. Safecard: a gigabit ips on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*. Hamburg, Germany.

DRUSCHEL, P., ABBOTT, M. B., PAGALS, M. A., AND PETERSON, L. L. 1993. Network subsystems design. *IEEE Network 7,* 4, 8–17.

DRUSCHEL, P. AND PETERSON, L. L. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*. 189–202.

DRUSCHEL, P., PETERSON, L. L., AND HUTCHINSON, N. C. 1992. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. 12th Int. Conf. on Distributed Computing Systems*. 512–520.

FEDOROVA, A., SELTZER, M., SMALL, C., AND NUSSBAUM, D. 2004. Throughput-oriented scheduling on chip multithreading systems. Tech. Rep. TR-17-04, Harvard University. August.

FIUCZYNSKI, M. E., MARTIN, R. P., OWA, T., AND BERSHAD, B. N. 1998. Spine: a safe programmable and integrated network environment. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*. ACM Press, New York, NY, USA, 7–12.

FRASER, K., H, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. 2004. Safe hardware access with the xen virtual machine monitor. In *Proceedings of OASIS 2004*.

FRÖHLICH, A. A. AND SCHRÖDER-PREIKSCHAT, W. 1999. Tailor-made operating systems for embedded parallel applications. In *Proc. 4th IPPS/SPDP Workshop on Embedded HPC Systems and Applications*.

GOVINDAN, R. AND ANDERSON, D. P. 1991. Scheduling and ipc mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 68–80.

HRUBY, T., VAN REEUWIJK, K., AND BOS, H. 2007. Ruler: high-speed packet matching and rewriting on npus. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, New York, NY, USA, 1–10.

HUTCHINSON, N. C. AND PETERSON, L. L. 1991. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17,* 1, 64–76.

ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of Eurosys'07*.

JACOBSON, V. AND FELDERMAN, B. 2006. A modest proposal to help speed up & scale up the linux networking stack. `http://www.linux.org.au/conf/2006/abstract8204.html?id=382`.

KARLIN, S. AND PETERSON, L. 2002. Vera: an extensible router architecture. *Computer Networks (Amsterdam, Netherlands: 1999) 38,* 3, 277–293.

KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. 2000. The click modular router. *ACM Transactions on Computer Systems 18,* 3, 263–297.

LEA, R., YOKOTE, Y., AND ITOH, J.-I. 1995. Adaptive operating system design using reflection. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. IEEE Computer Society, Washington, DC, USA, 95.

LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K. P., AND CONSTABLE, R. L. 1999. Building reliable, high-performance communication systems from components. In *Proc. of 17th ACM Symposium on Operating Systems Principles*. 80–92.

MCCANNE, S. AND JACOBSON, V. 1993. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX conference*. San Diego, Ca.

MCVOY, L. 1998. The splice I/O model. www.bitmover.com/lm/papers/splice.ps.

MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., PROEBSTING, T. A., AND HARTMAN, J. H. 1994. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*.

NAPI. Linux napi, or "new" network api. Documentation at `http://www.linuxfoundation.org/en/Net:NAPI`.

NGUYEN, T., CRISTEA, M., DE BRUIJN, W., AND BOS, H. 2004. Scalable network monitors for high-speed links: a bottom-up approach. In *Proceedings of IPOM'04*.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 2000. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems 18,* 1, 37–66.

PASQUALE, J., ANDERSON, E. W., AND MULLER, K. 1994. Container shipping: Operating system support for i/o-intensive applications. *IEEE Computer 27,* 3, 84–93.

POLYCHRONAKIS, M., ANAGNOSTAKIS, K. G., AND MARKATOS, E. P. 2007. Emulation-based detection of non-self-contained polymorphic shellcode. In *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007*. 87–106.

PRESOTTO, D. AND WINTERBOTTOM, P. 1993. The organization of networks in plan 9. In *Proceedings of the Winter 1993 USENIX Conference*. Usenix.

PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. 1995. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*.

REID, A., FLATT, M., STOLLER, L., LEPREAU, J., AND EIDE, E. 2000. Knit: component composition for systems software. In *OSDI'00*. USENIX Association, 24–24.

RITCHIE, D. M. 1984. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63,* 8, 1897–1910.

SCHAPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. 2008. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS'08)*.

SERMULINS, J., THIES, W., RABBAH, R., AND AMARASINGHE, S. 2005. Cache aware optimization of stream programs. *SIGPLAN Not. 40,* 7, 115–126.

SULLIVAN, M. AND HEYBEY, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proceedings of USENIX 98*. 13–24.

SUN MICROSYSTEMS, I. 2005. *STREAMS Programming Guide*. Sun Microsystems, Inc.

TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., AND MULLENDER, S. J. 1990. Experiences with the amoeba distributed operating system. *Commun. ACM 33,* 12, 46–63.

TRAW, C. B. S. AND SMITH, J. M. 1993. Hardware/software organization of a high-performance ATM host interface. *IEEE JSAC (Special Issue on High Speed Computer/Network Interfaces) 11,* 2, 240–253.

VEITCH, A. C. 1998. A dynamically reconfigurable and extensibe operating system. Ph.D. thesis, Univ. of British Columbia.

WELSH, M., CULLER, D. E., AND BREWER, E. A. 2001. Seda: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*. 230–243.

WULF, W. A. AND MCKEE, S. A. 1995. Hitting the memory wall: Implications of the obvious. *Computer Architecture News 23,* 1, 20–24.

Received Month Year; revised Month Year; accepted Month Year