PipesFS: Fast Linux I/O in the Unix Tradition

Willem de Bruijn
Vrije Universiteit Amsterdam
w.de.bruijn@few.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam and NICTA
h.bos@few.vu.nl

ABSTRACT

This paper presents PipesFS, an I/O architecture for Linux 2.6 that increases I/O throughput and adds support for heterogeneous parallel processors by (1) collapsing many I/O interfaces onto one: the Unix pipeline, (2) increasing pipe efficiency and (3) exploiting pipeline modularity to spread computation across all available processors.

PipesFS extends the pipeline model to kernel I/O and communicates with applications through a Linux virtual filesystem (VFS), where directory nodes represent operations and pipe nodes export live kernel data. Users can thus interact with kernel I/O through existing calls like `mkdir`, tools like `grep`, most languages and even shell scripts. To support performance critical tasks, PipesFS improves pipe throughput through copy, context switch and cache miss avoidance. To integrate heterogeneous processors (e.g., the Cell) it transparently moves operations to the most efficient type of core.

1. INTRODUCTION

With more than 8000 lines of code added, removed and modified daily [15], Linux is the largest single laboratory for operating system development. At its core is a conservative architecture, however, that was developed decades ago for time-sharing machines with relatively fast memory, slow processors and even slower networks. Since then, commodity computer architectures have grown more diverse and complex. Multicore CPUs are widespread and manycore [1], performance asymmetric [16], CPU/GPU hybrid (AMD Fusion, Intel Larrabee) and heterogeneous architectures (STI Cell, embedded SoCs) are in production or have been announced. Collectively, we refer to these innovations as *anycore* processors: parallel resources that are difficult to fully exploit with today's monolithic applications.

At the same time, the main bottleneck in I/O processing has moved to the memory system as peripheral I/O, such as graphics pipelines and networks, eclipsed CPU growth, while memory access latency failed to keep up (creating the "memory wall" [26]). To improve throughput, it is now essential to avoid all unnecessary memory operations. Inefficient I/O primitives exacerbate the effects of the memory wall by incurring unnecessary copying and context switching, and as a result of these cache misses.

Contribution. We revisit the Unix pipeline as a generic model for streaming I/O, but modify it to reduce overhead, extend it to integrate kernel processing and complement it with support for anycore execution. We link kernel and userspace processing through a virtual filesystem, PipesFS, that presents kernel operations as directories and live data as pipes. This solution avoids new interfaces and so unlocks all existing tools and languages for streaming I/O. In summary, PipesFS holds the following attractive characteristics:

1. **Unifies I/O processing** by making all kernel IO accessible through a virtual filesystem and all data through Unix pipes. Using this single primitive for all communication removes the need for domain-specific interfaces and enables the following two advantages system-wide.

2. **Increases I/O throughput** over stock Linux, by reimplementing pipes as shared memory structures and reducing copying, context switching and cache miss overhead. Observed throughput gains are 2x for a single pipe and up to 30x when copy avoidance is possible.

3. **Optimizes I/O on anycore** by automatically moving I/O filters to specialized cores (cryptographic logic, programmable co-processors) when available and by configuring such hardware below the OS interface, i.e., completely transparent to applications.

PipesFS is practical open source software, consisting of a Linux 2.6 kernel module, userspace library and device drivers. It can be downloaded from `http://netstreamline.org/`.

Outline. In the following section we introduce our application domain and explain why the Unix pipeline is effective in principle, but often shunned in practice. Then, we dedicate a section each to our three presented improvements: section 3 discusses the virtual filesystem, section 4 presents the pipe optimizations and section 5 the support for anycore processors. At the end of each section we summarize the differences with Linux and discuss to what extent the ideas can be integrated. In section 6 we discuss related work. Finally, we draw conclusions in section 7.

2. BACKGROUND

PipesFS targets streaming I/O applications, especially those that span multiple hardware tasks, e.g., because they involve IPC or spend a significant amount of time in kernel tasks. Applications that fit this description are Unix tools (which use pipes), network clients and servers (which require

significant protocol processing) and any application involving sequential access to peripheral hardware (cryptographic, graphics, video and audio processing).

Throughout the paper we use a single example application to clarify technical details and show practical utility. For this purpose we choose a webserver for hosting static content, because this application requires both application and kernel processing, is memory bound (as long as data resides in the disk-cache), makes use of all PipesFS's advanced features (multi-user pipes, splicing and hardware adaptation) and needs no further introduction.

2.1 Unix pipes

Natural growth of application performance with advances in processor technology has ended with the arrival of any-core architectures. The transition particularly impacts I/O applications, because the introduced parallelism intensifies contention on the already overloaded memory system. To make efficient use of modern computer architectures therefore requires a change in application design. The Unix development model (or 'philosophy' [9]) offers a proven alternative: build small *filter* programs that "do one thing and do it well" and connect these into larger, specific, applications through clean, simple interfaces – the quintessential example of which is the Unix pipe. Pipelines separate processing from transport logic, clearing the path for structural optimization of both. Compartmentalized logic can be mapped efficiently onto of a diverse set of computer architectures, and a uniform communication primitive ensures that transport optimizations (such as copy avoidance) are applied systemwide. In this paper we show a number of such optimizations to the basic pipe that together reduce I/O overhead by factors of magnitude (2 to 30x).

Unix pipes hold a few key advantages over alternative pipeline interfaces. Because the only shared interface is that of low-level Posix system calls, they can be accessed from all existing tools and virtually any programming language. Using pipes in a novel domain therefore immediately unlocks a large collection of tools (e.g., `grep` or `gzip`) for rapid adaptation of applications. Second, because pipes are so widespread (just not for high-speed I/O), the developer learning curve is less intimidating than for yet another interface. Program structure can be more easily comprehended and filters can be debugged and optimized independently. Third, the system scheduler can adapt scheduling of pipelined tasks to fit local hardware. It can exploit the explicit structure and modularity of pipelines to co-schedule filters so that CPU utilization and cache hitrate are maximized on the local combination of cores and caches, in ways that developers cannot because they lack this hardware information at compile time. Most importantly, though, pipes are to be preferred from an architectural point of view: as the basis of the Unix development model, they have frequently shown to improve code reusability and maintainability [9].

2.2 Linux practice

Why, then, do developers avoid pipes? First, compared with local pointer arithmetic within a process, pipe based IPC *as currently implemented* incurs considerable copying, task switching and cache miss overhead. To avoid such non-functional I/O overhead, developers are forced to replace modular designs based on these clean primitives with monolithic tasks that employ explicit pointer handling and domain-specific interfaces – duplicating code, increasing complexity, and complicating scheduling. Indeed, in practice, performance critical applications in Linux generally eschew pipes, for instance for media processing (gstreamer, alsa) and networking (netfilter). Second, Unix pipes are a userspace primitive only, but especially I/O applications require significant kernel processing, because they need to access protected hardware resources such as graphics processors and network interfaces. Communication between the two environments is an important source of transport overhead. As a result, incompatible domain-specific interfaces have been developed to reduce cost for selected applications (e.g., `sendfile`, Linux netlink).

With PipesFS we avoid both pitfalls. We improve end-to-end I/O performance by modifying Unix pipes to better fit modern hardware: exchange pointers to avoid copying, buffer data to avoid task switching, and adjust working-sets to fit cache-sizes. Contrary to application logic, we enable such optimizations systemwide and cleanly, because we execute them *behind the pipe interface*. To integrate kernel processing, PipesFS structures kernel tasks as yet another pipeline and enables generic, efficient communication between kernel and application tasks by exporting all kernel streams to userspace through fast Unix pipes. It renders *all* kernel pipes accessible as well as the interconnecting pipeline structure, through a virtual filesystem.

3. A VIRTUAL FILESYSTEM FOR I/O

PipesFS reuses the filesystem namespace, which is a generic digraph, to present kernel I/O. Like FIFOs, filesystem-backed pipes can be resolved by all applications. PipesFS goes one step further and exploits the private namespace to visualize kernel pipelines as directory hierarchies. Doing so allows applications to control kernel I/O through existing system calls, such as `mkdir`, avoiding the introduction of yet another API. Likewise, PipesFS makes all data streams between kernel filters accessible from userspace as Unix pipes, so that applications can access and modify network, audio and video streams without having to resort to domain-specific interfaces. As a result, it becomes trivial, for instance, to log all requests to our webserver to a compressed archive. For PipesFS mounted at `/pipes` the shell command

```
cat /pipes/.../http/get/all | compress > log.Z
```

suffices, whereby we abbreviated the path for clarity. This command 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. This pipeline continues until we reach data sources, in this case a network interface.

Modifying a pipeline, for instance to insert a layer 7 protocol filter after a vulnerability in our webserver is discovered, is as simple. Moreover, the complex protocol 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 will return to this example and discuss its performance implications shortly. First, we introduce the filesystem and its components (directories, pipes and files).

3.1 Design

PipesFS is a Linux virtual filesystem with support for nesting, symbolic links and the following special properties: directories represent kernel filters, pipes present filter in- and output and a filter's output is automatically piped to the input of all its children. In other words, data flows downwards from the root to the leaves.

Directory nodes as filters. Each directory in PipesFS represents an active filter running in the kernel. PipesFS comes bundled with more than 20 kernel filters – ranging from protocol filters to device driver interfaces – and a runtime system that controls I/O between filters. The implementation of kernel filters is similar to that of `netfilter` elements and indeed, we could easily integrate those, but at present we only support homegrown filters.

Pipe nodes as output. As filters deal with sequential I/O, their data can be represented as pipes. PipesFS exports each filter's output through a pipe childnode in the filter directory. In its most basic form, all output from a filter `/pipes/X` can be accessed from the pipe `/pipes/X/all`. Deviating from pure bitpipes, PipesFS pipes can be both byte and record oriented. In the latter case, calls to `read` and `write` return at most the number of bytes in a single record. This behavior is allowed by Posix standards and is required to integrate discrete (e.g., network packet) streams.

Because many protocol filters classify contents (e.g., by IP protocol field), our kernel filters present a 16 bit classification number along with each block of data which can be used in domain-specific ways. For instance, the IP protocol filter uses it to differentiate between transport-layer protocols. The runtime system makes each logical substream available through a private pipe in the subdirectory `F/N`, for each N between 0 and 65535. Additionally, the runtime system can resolve numbers to names, e.g., to replace the lookup for TCP traffic `ip/6` with the more friendly `ip/tcp`. To make use of this feature, filters must (statically) register a lookup table. Symbolic links are then automatically generated.

File nodes as control. All but the most trivial filters take parameters. In PipesFS, communication between filters and applications takes place through files. In principle, all files in a directory are randomly-accessible (as opposed to pipes) memory regions. Writable control files set parameters, read-only files are used to communicate metadata to the application. Small files copy data during each call, large files are backed by the same shared memory regions as pipes (to which we return in the following section) to reduce overhead of high-throughput communication. Additionally, filters can manually add metadata pipes, to replace such in-band metadata as socket timestamps.

To reduce clutter, the runtime system generates files for all the filter's configuration options prepended with a dot. These names cannot clash with filters, because those may only have alphanumerical characters in their name.

Nested structures as tasks. To create meaningful applications, users connect filters into composite graphs through use of the `mkdir`, `rmdir`, `link` and related Linux system calls. PipesFS also implements symbolic links, to expand the FS graph type from a tree into a digraph. Whenever a write call is issued on a pipe, either from the filter or from a userspace application, the filesystem intercepts the call and automatically calls all children with the same block of data. This simple recursive algorithm enables data to reach all dependent nodes in the graph. Because a cycle leads to an infinite processing loop for nearly all combinations of filters, these must be avoided. PipesFS does not currently detect cycles.

Root. The filesystem root node, as only exception to the rule, does not correspond to a filter and does not generate any output. Instead, for useful processing, children of the rootnode must generate output without input. Device drivers are examples of this. Besides these children, the root has a single file, `.available`, which lists all the available filter implementations. `mkdir` will only succeed with names from this list.

3.2 Sockets

Like Plan9, we aim to replace domain-specific interfaces with operations on files. One widely-used such interface is the socket. Alternative network interfaces, even modular ones, are as old as sockets themselves [24], but for a long time were not as efficient. At present, however, we have a number of reasons to move away from sockets, aside from the obvious goal of collapsing APIs. First, the interface is complex with 5 calls each for reading and writing data, various types of in-band metadata messages and an exceptional call, `sendfile`, for sending file contents. Second, the interface shields all kernel I/O from applications, so that network operations (e.g., packet filtering) require additional interfaces and must operate in kernelspace. Third, unless the kernel implements sockets in a modular fashion – duplicating Unix pipe functionality –, load balancing network processing across parallel processors is non-trivial. Finally, the cost incurred by composite network stacks is no longer significant [2]; on the contrary, we will show in the next two sections, 4 and 5, how separating transport from processing logic benefits efficiency by enabling global optimizations to both.

PipesFS replaces socket-specific data access calls (like `send` and `recvmsg`) with basic reading from and writing to pipes, whereby the location of the pipe identifies the socket. The actual path for sockets is long, consisting of 8 filters for the reception end, but users are easily shielded from this complexity through symbolic links (e.g., `/pipes/sockets/tcp/N`). Similar to the *udev* device file system, a userspace daemon could be written set up such links automatically. Additionally, we replace the control calls `connect` and `bind` with filesystem operations on control files. Currently not supported are waiting on multiple sockets and connection handling, but both can be implemented using Linux's file notification infrastructure `inotify`. In-band messaging is replaced with separate metadata pipes.

3.3 Usage

PipesFS filesystem nodes are modified like any other, using Posix system calls. We discuss both configuration and access calls.

Configuration. Filter instances – a combination of logic and state – are created using `mkdir`. During creation, a filter's input is immediate connected to the output of its parent. If parameters need to be set prior to connection, the filter must be created directly below the rootnode and moved after initialization. Once created, filters can be freely moved and copied. Moving a filter (executing the `rename` system call on the directory node) closes the input pipe (if any) and opens the new parent's output pipe. Copying only involves a `mkdir` call and is therefore similar to creation of a new node. Because a recursive copy operation is not atomic, input should also be temporarily severed to guarantee consistent state. Moving the source node to the root level for the duration of the operation achieves this, but must be manually executed.

Read and write access. PipesFS not only exposes the kernel I/O configuration, it also makes all live I/O streams accessible from userspace. As the logging example demonstrated, reading data requires the same actions as reading from regular pipes. Modifying the contents of an I/O stream is more involved. To achieve this, a user must sever the stream – again by moving the child directory to the root level – and manually forward data from the (former) parent to the child. This entails reading from the output pipe, applying any required transformation to this data and then writing it to the input buffer. At this point we continue with the layer 7 (i.e., deep inspection) protocol filter example that we introduced at the start of this section. Traditionally, to filter at this level, all network data must be intercepted and parsed up to the protocol level, because no hooks in the regular socket handling code exist to attach to. With transparent I/O, not only can we move processing out of the kernel (if we wish, this is not a requirement), protocol processing is not unnecessarily duplicated. 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):

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

3.4 Linux integration

PipesFS has a Linux 2.6 virtual filesystem implementation that can be downloaded as part of the larger *Streamline* package. The complete package implements a reconfigurable I/O architecture for heterogeneous and distributed systems; PipesFS exposes the most practical features in such a way that it naturally fits the Unix model and Linux OS. In its current implementation, PipesFS still has strong dependencies on other parts of Streamline – specifically on the filter implementations and fast pipes. The first can be replaced, at least in part, by Linux `netfilter` and `crypto` elements and fast pipes can initially be eschewed in favor of their existing copy-based implementations. For high throughput, a facility similar to our fast (shared memory-based) pipes is required. We now turn our attention to the implementation of these pipes.

4. FASTER PIPES

Clear compartmentalization of I/O into userspace and kernel and further into independent processes and kernel subsystems as found in Linux today limits throughput, by incurring non-essential copying, context switching (both task-switching and mode transitioning) and as a result cache pollution. This is particularly true for PipesFS, which switches between compartments frequently. For this reason we replace the common pipe with a *semantically identical* structure based on shared memory. We have previously presented these structures, Beltway Buffers, in depth [3]. Here, we limit discussion to those parts relevant to PipesFS. We will show that the Beltway Buffers used in PipesFS incorporate many optimizations that significantly reduce overhead. Because these optimizations are hidden behind a uniform POSIX file interface that is used throughout the system, these optimizations are not only transparent to the user, but also automatically applied system-wide.

4.1 Shared buffers

Beltway replaces strict data-separation with a system-wide I/O architecture built from virtually contiguous shared memory areas, or *I/O regions*. Regions are statically mapped into the address spaces of all interested parties: userspace applications, kernel tasks and occasionally peripheral devices (in which case they may have to be physically contiguous as well). Data producers write data into a region once, from where all consumer access the data locally, i.e., without context switches. Because the data access pattern in streaming I/O is largely sequential and in principle unbounded, data is stored as a fixed-length window over a stream. A ring-buffer is a fixed-length buffer with sequential ordering that wraps around its container. Shared ring-buffers offer three advantages over competing I/O architectures: (1) they remove all per-block allocation overhead, (2) they remove most per-block transport cost, where others must copy data or modify a VM mapping for each block and (3) they are cheap to access, because they avoid dynamically allocated linked structures and keep data packed closely, at the benefit of locality of reference.

Buffers export the Posix I/O interface (with `open`, `read`, etc. calls). What sets Beltway apart is that these calls are implemented everywhere using local function calls. In other words, no mode transitions are needed to access streaming I/O. Even in userspace, what are traditionally seen as system calls are handled in the context of the application, removing potentially many — because one for each block — mode transitions. Direct access is a potential breach of process isolation, but we will return to this point in section 4.6.

Multiple clients. Beltway Buffers allows parallel consumer access, but serializes producers. Multi-consumer access replaces copying. For example, our webserver and logging tool want to read the same data, so they can share access to the same buffer. Each client that opens a buffer (using the `open` call) gets a private view on the buffer, similar to how each file `open` presents a private file offset. No such reason exists for multiple producers, because all writes occur at the head of the ring. In Beltway, each client has at least three I/O regions: one private region holding the client's read offset and other private metadata such as open flags, one shared region holding the shared write offset (to which consumers require

access for synchronization) and metadata such as buffer implementation type and one large shared region holding the actual data. Splitting the buffer into disjoint regions has the additional advantage that it trivially avoid cache conflicts due to shared cachelines. The idea is similar to that of Van Jacobson's netchannels [12].

Multiple rings. Beltway integrates all buffers into a *copy avoidance network*. Ideally, each I/O stream would have its own ring-buffer; this is infeasible, however, as data would have to be copied between streams frequently and buffer setup/teardown cost would dominate tasks with many streams (such as webservers). Beltway therefore moves all data into as few *shared* buffers as possible. Data partitioning is only performed for one of four reasons: security, multiprocessing, distributed memory and modification. Security in this context concerns data isolation (e.g., for privacy). We argue that much more sharing is possible than is currently permitted in Linux. For instance, on a dedicated server, userspace applications can be granted direct access the network reception buffer through shared memory. It may even be quite acceptable to let multiple independent applications access this buffer (read-only), for instance when the physical network is also unprotected. Multiprocessing performance is improved when cache conflicts from false data dependencies are avoided. In the presence of multiple memory areas (e.g., on peripheral devices or NUMA nodes), explicit copying of data once is cheaper than recurring zero-copy access across high-latency links. Finally, when applications want to modify data and concurrent claims exist on a stream, a private copy must be made to avoid data corruption.

4.2 Indirection

To further reduce copying, while allowing private *logical* streams, Beltway introduces transparent indirection: a software-based (and thus unprotected) virtual memory layer that eschews hardware isolation to minimize mode switching overhead and, as a result, is not bound to page sizes. Shared memory by itself only removes copies across kernel subsystems and the application binary interface (ABI); indirection removes additional copies unrelated to such boundaries. Such copies occur when reconstructing a stream from disjoint blocks or when instructed to transfer data from one file descriptor to another. In the first case, indirection acts purely as a virtual memory layer, arranging pointers to present a contiguous view when there is none. In PipesFS, we use this feature to avoid copying near-identical data between parent and child pipes (e.g., to reconstruct TCP streams from IP packets for our webserver). The second case is known as splicing: copy-free movement of data through the system, which we exploit to cheaply move data from one pipe through a userspace process (such as the layer-7 filter presented earlier) to the next.

Beltway fuses independent data buffers, or *DBufs*, into an integrated architecture through indirect buffers, or *IBufs*. IBufs are also ring-buffers, but instead of data, they contain small indices that point into *DBufs*. IBufs export the exact same interface as DBufs, so that clients can remain unaware of whether indirection is used. IBufs differ from DBufs in that instead of returning their buffer contents on a `read` they resolve the referenced DBuf and return a data block from within that buffer. Beltway silently handles "buffer faults", situations where a DBuf is not mapped into the local memory protection domain, by setting up a long-lived mapping. After a mapping is made on the first 'miss', we incur no more allocation or VM overhead at runtime. During a `write`, IBufs both write the index and perform a write-through operation into a DBuf that acts as backing store.

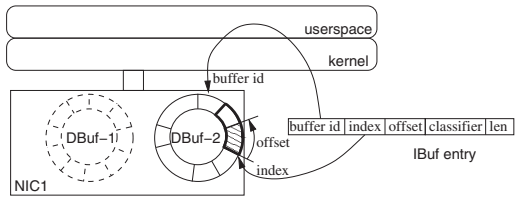

Figure 1: An index points into a DBuf

Indirection is an effective copy-avoidance concept. It is usually implemented using pointers embedded in queues or heavyweight datastructures (e.g., Linux's `sk_buff` network packet structure). IBufs have a few advantages over explicit pointer access. They centralize indirection, which enables clients to communicate using the safe and clean Posix file interface instead of through error-prone pointer handling. More importantly, centralization also enables the I/O architecture to perform transfer optimizations "under the hood", automatically, for all applications. Furthermore, IBufs enable cross-space indirection and copy-avoidance, whereas pointers are limited to a single address space. IBufs pack indices close together, to maximize cache hit-rate. Finally, IBufs speed up processing by allowing some filters to operate only on the (cached) index, never touching the data at all.

The index format is important. Indices should be as small as possible to compress IBufs (and increase cache hit-rate), yet be expressive enough to be usable for all applications. The chosen format, shown in Figure 1, combines an address-space agnostic "rich pointer" into a DBuf with a single metadata field, known as the classifier. A rich pointer is a 4-tuple of global DBuf identifier, slot offset within the buffer (if the implementation has fixed-size slots), block offset within the slot and length. Clients are free to use the classifier for domain-specific communication. For instance, a layer-7 protocol filter uses it to signal the threat level.

4.3 Splicing

Splicing is the copy-free movement of data from one stream into another. McVoy [17] advocated the introduction of a `splice` system call to Linux to allow copy-free movement of data within the kernel. This call is now part of the mainline Linux kernel, together with related `vmsplice` and `tee` calls. Beltway implements splicing without the need for new system calls and can splice data throughout the system, instead of limiting itself to userspace control of kernel memory. On the other hand, Beltway only allows splicing to IBufs, because splicing is implemented by writing to an IBuf without performing the write-through to a DBuf. The optimization is performed below the Posix `read` and `write` calls and thus requires no client support, i.e., it is a transparent, backwards compatible optimization. For every write operation of sufficient length to an IBuf, the passed pointer is compared to the address ranges of all known DBufs. If one matches, a rich pointer is computed from the DBuf implementation and only this index is written. Because address range matching

comes at a cost, `write` reverts to non-splicing mode if the hit-rate drops below 80% (i.e., when misses are not rare).

Besides basic splicing, Beltway supports splicing of application buffers. One of the drawbacks in terms of performance of the Posix interface is that it imposes copy semantics. For applications that mainly move data from one buffer to another, such as network file servers (e.g., our example webserver) or PipesFS filters, copying can be the main – and unnecessary – bottleneck. To splice application buffers, Beltway keeps an LRU list with pairs of application buffer pointers as passed to `read` and the accompanying DBufs from which data was requested. In the `write` call it compares the then passed application buffer pointer not just to DBuf address ranges, but also to this list. If one matches, data is spliced immediately from the DBuf. This method is clearly not safe, as it fails to track whether application buffers were changed in the meantime. Beltway supports two solutions: either enable the optimization manually through a flag passed to the `open` call, or incur a mode-transition once and revoke write permissions to the application buffer. Then, if a page-fault occurs due to writing, data is copied lazily, permissions are reinstated and the buffer is removed from the LRU list. To avoid the high cost of page-faulting, the optimization is then also disabled for future `read` calls from this process. Splicing achieves copy reductions like the `sendfile` system call, but is more generic and transparent. Our unmodified webserver, for example, benefits from splicing.

4.4 Cache optimization

To maximize I/O throughput, working-sets must be scaled to an appropriate cache-size (usually L2 [7]). Ring-buffers lend themselves well to cache-tuning, because they are fixed-size, contiguous segments, which makes their hit-rate fairly predictable. On the other hand, because access within a ring is sequential, even moving one element (byte, slot, block) beyond a cache size will result in consistent misses, as each next element will be the last accessed and therefore first to be evicted. We observed a sharp four-fold decrease in throughput by only barely exceeding the L2 size [3].

For this reason, Beltway integrates two methods for automatic attuning to cache-size. Both are based on runtime resizing, but their implementation differs. The true resizing buffer takes a cue from rehashing: it allocates a memory area of 2 times its current size, if a pressure value exceeds either a high watermark when the producer wraps. Instead of wrapping, the producer then continues writing in the new region. Similarly, if the pressure value drops below a low watermark, it allocates and writes into a buffer of half the original size. Pressure is a weighted moving average of the distance between producer and consumer(s). The other method, pseudo-resizing, does not allocate a new block but moves an internal boundary – similar to how a deck of cards is cut – and releases the unused portion, e.g., to the diskcache. Which method is appropriate for a given situation depends on the number of expected resizing operations. True resizing is a more expensive operation, but conserves more memory in the end.

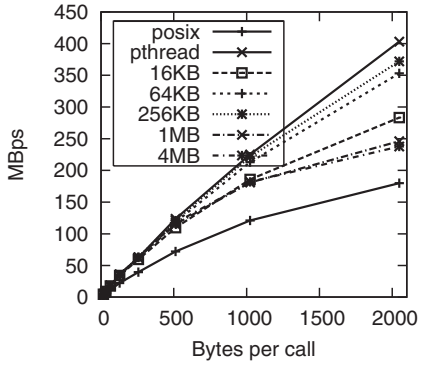

Figure 2: Evaluation of pipes

4.5 Selective buffering

Even though all streams are accessible in principle, in practice PipesFS avoids buffering for the many unused pipes. Only when a pipe is explicitly opened by an application is data written to a buffer. Conversely, buffering stops and the IO region is destroyed when the last user closes its end. In the fast common case, pointer forwarding replaces buffering and filter execution takes the form of direct function calling.

4.6 Security

Beltway offers the same level of isolation as traditional Posix implementations. Like files, buffers have a personal access control policy with different rules for user, group and others. If required, one can also restrict sharing to a single process. Access control is performed once per buffer and memory protection domain pair: when a buffer is initially mapped into the memory protection space. We discuss access control in more detail elsewhere [3].

Shared memory I/O allows us to selectively relax data isolation where doing so is safe and improves performance. Traditional copy-based system calls enforce isolation stringently and consistently. Beltway Buffers, on the other hand, can reconsider isolation utility on a case-by-case basis when the interface does not prescribe isolation. Sharing data is a security concern, but also advantageous to performance as it save many copies and context switches. In many situations in which isolation is now enforced, it is done so unnecessarily. Strict isolation is necessary is multiuser environments, but one can question privacy concerns on dedicated workstations and servers. Moreover, any data that is freely accessible outside of the confines of the I/O architecture, such as that stored in globally readable files or transferred over public network links need not be protected by the kernel. For example, if our webserver executes on a dedicated server, mapping the reception DBuf into the server process space causes no direct privacy concerns; neither does mapping the transmission descriptor rings.

4.7 Performance

The presented architecture reduces copying, context switching and cache pollution by switching to shared memory and indirection universally and by organizing memory to maximize cache hit-rate. Pipes that do not introduce data (unlike, for example, device drivers) are implemented using IBufs and both these and the underlying DBufs are mapped into all applicable memory protection domains. Beltway

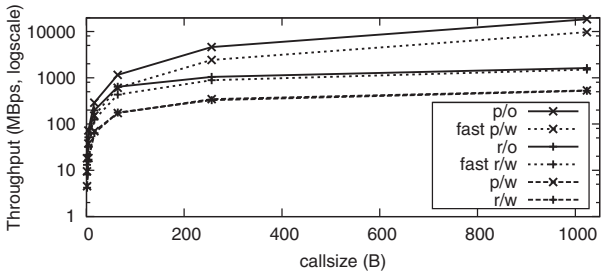

Figure 3: Evaluation of copy-avoidance

increases cache hit-rate by minimizing size of and closely packing metadata structures, by avoiding false data dependencies on multiprocessors and by scaling memory areas to cache size.

We summarize the results of comparing Beltway directly to stock Linux 2.6.19 [3]. Observed throughput improvements are 2x for pipes, mainly due to a reduction in task switching, up to 10x for packet capture, for the same reason, and 3x to 30x for copy avoidance at network packet sizes. Figure 2 shows pipe throughput. Linux (with 64kB buffer) is slowest, a pthread application that uses explicit shared memory instead of pipes and threads instead of processes is more than twice as fast and a well tuned Beltway Buffer is within 10% of this result. Figure 3 shows the impact of copy-avoidance: the two slowest method use copying both on `read` and `write`, the two middle elements are approximately 3x as efficient by only incurring a copy on `read` and the two fastest methods, which only access indices, and are again a decimal order of magnitude more efficient.

4.8 Linux integration

Shared memory I/O, indirection, splicing and buffer attuning to cache size improves I/O throughput independently of the I/O VFS. Indeed, Linux already incorporates some of these optimizations, but only in selected situations. The `PF_PACKET` socket can setup a large, contiguous shared ringbuffer similar to a single Beltway Buffer, recent Linux kernels export a splice interface and Glibc 2.3 transparently switches to memory mapped file I/O when doing so increases sequential `fread` performance [5]. Beltway Buffers makes such optimizations more generally available, but can be introduced on a case-by-case basis – for instance, starting as a Unix pipe implementation.

5. ANYCORE PROCESSING

Application portability has long been a strong point of Unix. For a large part, this is due to the fact that the interface is hardware independent: instead of exposing physical device details, the OS communicates through abstract concepts, such as processes, files *and pipes*.

Anycore architectures pose three challenges to writing efficient portable applications: parallelism, heterogeneity and diversity. Hardware is increasingly parallel, but few applications scale to exploit this. Those that do commonly use multi-threading or multi-processing based on `clone` or `fork` and therefore cannot incorporate heterogeneous cores. Moreover, processor number and kind differ from one machine to the next, rendering architectural optimization at compile-time untenable. Instead, structural decisions have to be made at runtime, based on locally available CPU, memory and peripheral hardware availability.

In principle, pipelines can solve both the parallelization and heterogeneity issue for streaming I/O applications, at runtime. Composite pipelines map trivially onto diverse sets of distributed resources and because of the simple interface between filters, can combine heterogeneous elements. We have built I/O applications for anycore processors using Streamline, the reconfigurable architecture on top of PipesFS that we briefly introduced in section 3.4. Like similar systems [8, 13, 22]) we initially targeted network applications and for this purpose integrated two generations of network processors, the Intel IXP1200 and IXP2xx0. The IXPs have one CPU complemented by 6 or more simple RISC processors that execute without OS. Having applications automatically make use of such hardware when available requires runtime resource discovery, selection, allocation and configuration (such as firmware loading). Streamline automates these tasks behind the PipesFS interface.

5.1 Heterogeneity

Current operating systems do not integrate heterogeneous hardware transparently (i.e., behind the ABI), but expose all physical details directly to the application. Moreover, they lack even a generic interface for exposing these details: device drivers export their own, mutually incompatible, versions. Linux support for the Cell BE and Intel IXP2xx0 is representative. For both processors the OS exports mainly a firmware loading interface, but the two versions are incompatible: one is a VFS, the other uses `sysfs`.

Streamline takes a different approach and presents the abstract, task-oriented, PipesFS interface. Whereas PipesFS as discussed in section 3 maps requests immediately onto kernel filters and fails if no implementation exists, Streamline adds a layer of indirection: it models the computer architecture as a digraph of *execution spaces* (OS kernel tasks, IXP co-processors, etc.) and for each type of space maintains a library of filter implementations. For each filter request, Streamline searches through the set of libraries, selects an implementation and allocates resources of the required type. A greedy selection algorithm gives results acceptable for most situations with little overhead. The current algorithm prioritizes spaces as follows: prefer a dedicated resource (such as the IXP2850's cryptographic engine), else a special-purpose programmable core (such as the IXP's RISC processors), else run as part of the OS controlled pipeline. We call the general guideline "push processing down".

The process is simple in principle, but substantial support logic is required to hide all hardware details, especially for configuration. Example tasks are passing an initialization vector (IV) to the cryptographic unit, loading firmware onto co-processors or even compiling high-level languages into firmware. For this reason Streamline also supports *metafilters*: operations that are selected by the greedy algorithm even before dedicated cores, and perform a support operation that expands the algorithm's search space. For example, the metafilter `aes` prepares the IXP2850 cryptographic unit, rewrites the original request `aes` into `ixpcrypt` and reinserts the request. As a result of request reinsertion, metafilter re-

quests are stackable. On a programmable co-processor without cryptographic unit, such as the Intel IXP2400, the `aes` filter first compiles into object code and then loads onto the co-processor.

A separate case study discusses our implementation of regular expressions on an IXP2400 [10]. This work presents the network processor to Linux as a 'normal' Ethernet card with additional configuration options (the regex rules). On regular NICs, Streamline must execute `regex` filters in the kernel, but if this card is encountered, it can offload both the network and regular expression functionality to the card. After configuring the device, the filter becomes part of the PipesFS structure and users may connect it to other filters by calling `mkdir`, and access its in- and output through UNIX pipes, as if it were running on the host.

5.2 Parallelism

Pipelining also aids I/O application scheduling on parallel homogeneous hardware. As we discussed in the introduction, I/O applications are commonly memory-bound. Pressure on the memory system can be reduced by maximizing cache efficiency. This, in turn, is achieved by executing tasks that access the same data near in time and space: in the same timeslots on processors that share an L2 or L3 cache. Pipelines facilitate cache-aware scheduling because they render the data dependencies between filters explicit. We are currently building an *assembly line* scheduler for PipesFS that optimizes I/O throughput (not CPU utilization) by keeping filters closely synchronized in terms of throughput and by optimizing the mapping of adjoining filters onto successive timeslots and neighboring CPUs for diverse numbers of cores [4].

5.3 Linux integration

PipesFS and the Streamline reconfiguration layer are both implemented as modifications to Linux 2.6, but especially the latter is far removed from current Linux practice. Still, some lessons can be backported quite easily. Linux already exports task-oriented interfaces from its device filesystem. For instance, `/dev/random` is a common interface for obtaining pseudo-random numbers and `/dev/audio` for reading raw digital audio. Both present a streaming interface similar Unix pipes. This practice can be extended to include common cryptographic (e.g., `aes`, `hash128`), media (`mp3dec`), network (`regex`, `csum`) and other (`compress`) streaming operations that applications prefer to offload because they are computationally intensive and fairly independent of other code.

When the operating system is responsible for these critical operations, it can replace kernel implementations with faster alternatives when available. To support programmable resources, the OS must maintain a library of filter implementations for each local architecture. The implementation can be similar to `/lib/firmware`, only specialized for each processor type. Streamline's greedy algorithm then reverts to a keyword search through the various firmware directories.

Moving nodes from the device filesystem to PipesFS, finally, combines hardware offload with cache-aware scheduling on anycore architectures and with the more general advantages discussed in section 3. It enables communication between filters that execute on fast co-processors (e.g., the Cell SPUs) without interference from application logic scheduled on a CPU.

6. RELATED WORK

PipesFS is not the first virtual filesystem for I/O. Linux implements pipes through `pipefs`. That is not a generic filesystem, however, as it cannot be mounted and makes no use of nesting. Plan9 introduces a VFS for network streams to reach its design goal that "everything is a file" [23]. Like PipesFS, this replaces sockets with filesystem operations (and both avoid introducing network namespaces [21]), but only PipesFS expands into a generic I/O pipeline through nesting.

With other interfaces, pipelines have been proposed before for performance critical I/O processing. Streams [24] is the first practical example – and still widely deployed. The interface is not nearly as popular as sockets, however, at least in part due to lower performance. The x-Kernel [11], Scout [18] and the Click router [14] are more efficient descendants. Scout is even available as a linux patch [2]. What sets PipesFS apart is that it cleanly combines an existing I/O interface (Unix pipes) and operating system (Linux) while increasing performance over the standard configuration.

PipesFS implements copy avoidance, similar to container shipping [20], fbufs [6] and IO-Lite [19]. All approaches reduce copying, but PipesFS additionally reduces context switch and cache miss overhead by moving to large shared memory areas. Our buffer implementation is based on asynchronous rings, like netchannels [12] and we employ signal throttling, similar to clocked interrupts [25] and Linux's NAPI network device interface. Unlike IO-Lite, PipesFS cannot splice to the diskcache. On the other hand, it supports more generic application splicing [17], like recent Linux kernels. Unlike the Linux implementation, PipesFS does not introduce a new API.

7. CONCLUSIONS

We have presented PipesFS, a Linux virtual filesystem that structures all kernel I/O as a pipeline, presents this to applications as a nested directory structure and makes all live data accessible through Unix pipes. Performing all I/O through this single primitive improves code reusability and maintainability and benefits application performance, by introducing transport and processing optimizations systemwide. PipesFS increases pipe throughput by between 2x and 30x through copy, context switch and cache miss avoidance and demonstrates that Unix pipelines are a simple, yet powerful, abstraction for building applications that scale from uniprocessor to anycore architectures.

Acknowledgments

We would like to thank Patrick G. Bridges for commenting on earlier versions of this paper.

8. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view

from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[2] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. Silk: Scout paths in the linux kernel. Technical report, Uppsala University, 2002.

[3] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *INFOCOM 2008*, 2008.

[4] W. de Bruijn and H. Bos. Model-t: rethinking the os for terabit speeds. In *Workshop on high-speed networks (HSN 2008), Co-located with INFOCOM*, 2008.

[5] U. Drepper. Gnu c library version 2.3. In *UKUUG Linux Developers' Conference*, 2002.

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

[7] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Harvard University, August 2004.

[8] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. 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*, pages 7–12, New York, NY, USA, 1998. ACM Press.

[9] M. Gancarz. *The UNIX philosophy*. Digital Press, Newton, MA, USA, 1995.

[10] T. Hruby, K. van Reeuwijk, and H. Bos. 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*, pages 1–10, New York, NY, USA, 2007. ACM.

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

[12] V. Jacobson and B. Felderman. A modest proposal to help speed up & scale up the linux networking stack. `http://www.linux.org.au/conf/2006/abstract8204.html?id=382`, 2006.

[13] S. Karlin and L. Peterson. Vera: an extensible router architecture. *Computer Networks (Amsterdam, Netherlands: 1999)*, 38(3):277–293, 2002.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[15] G. Kroah-Hartman. Lkml discussion on rate of kernel changes, Feb 2008. `http://kerneltrap.org/mailarchive/linux-kernel/2008/2/2/700024`.

[16] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of Supercomputing'07*, 2007.

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

[18] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*, 1994.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[20] J. Pasquale, E. W. Anderson, and K. Muller. Container shipping: Operating system support for i/o-intensive applications. *IEEE Computer*, 27(3):84–93, 1994.

[21] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2), 1993.

[22] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *INFOCOM*, pages 67–76, 2001.

[23] D. Presotto and P. Winterbottom. The organization of networks in Plan 9. In *USENIX Association. Proceedings of the Winter 1993 USENIX Conference*, pages 271–280 (of x + 530), Berkeley, CA, USA, 1993. USENIX.

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

[25] C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance ATM host interface. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):240–253, 1993.

[26] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.