KStreams: Kernel Support for Efficient End-to-End Data Streaming

Jiantao Kong and Karsten Schwan
College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{jiantao,schwan}@cc.gatech.edu

Abstract

Technology advances are enabling increasingly data-intensive applications, ranging from peer-to-peer file sharing, to multimedia, to remote graphics and data visualization. One outcome is the considerable memory pressure imposed on the machines involved, caused by application-specific data movements and by repeated crossings of user/kernel boundaries. We address this problem with a novel system service, termed KStreams, a general facility for manipulating data without using intermediate buffers when it moves across multiple kernel objects, like files or sockets. KStreams may be used to implement kernel-level services that range from application-specific implementations of *sendfile* commands, to data mirroring or proxy functions, to fast path data conversions and transformations for data streaming. The KStreams API permits individual applications to define fast path operations, which will then execute at kernel level and if desired, without further application involvement. By placing application-specific data manipulations into data movement fast paths, user/kernel boundary crossings are avoided. By operating on data streams 'in-flight', data buffering is made unnecessary, thereby further reducing the memory pressure imposed on machines.

KStreams is implemented on Linux kernel version 2.4.22. Its evaluation uses data-intensive tasks performed in conjunction with modern web services, such as proxy functions, remote media streaming, data visualization, etc. Initial experiences with the KStreams implementation are encouraging. Fast path data transformation via KStreams results in increased throughput of 20-50% compared to user-level data manipulations. Future work with KStreams uses it with complex multi-machine web services, evaluated with representative user loads and applications.

1 Introduction

Applications like web and transaction services, publish/subscribe, or remote collaboration are becoming increasingly common. One outcome is a growing demand for services that involve frequent or large-scale data streaming. Supporting web requests, such streaming services range from protocol-level functionality like TCP routing, TCP handoff, or IP packet rewriting [3, 5, 6], to proxy forwarding and stack splicing [4, 8], to in kernel data paths [1] and data streaming system calls like *sendfile*, to kernel-resident web servers like khttpd [7, 12] that are able to load-balance dynamic requests across multiple remote machines. For highly reliable servers, data replication or mirroring have been shown useful [9, 16]. To handle multi-media applications or remote data visualization, kernel-resident data filtering [11] can reduce the memory loads imposed by such large-data applications, and QoS management actions include the packet scheduling or traffic differentiation needed by many such codes [13, 14].

Our research is creating and evaluating a small set of kernel abstractions that support the efficient, kernel-level representation of data streams. The KStreams kernel service is a 'thin' layer on top of the kernel's network and file subsystems. KStreams abstracts files and sockets into unified data objects that may be manipulated by application-specific operations. It provides an API to define such operations, termed stream handlers, to specify the objects that represent incoming data, the type of incoming data, and the relationships among multiple such objects (if data arrives on multiple incoming points). It also specifies the outgoing data objects and their relationships (when there are multiple ones). Examples in our work include (1) a data stream that arrives on multiple incoming sockets and is forwarded to multiple outgoing sockets in order to load-balance the requests it contains across different servers [27] and (2) a single incoming data stream mirrored to multiple remote sites to maintain reliability or uptime properties for transactional applications [9]. Other examples focus on data stream manipulations, such as data downsampling [28], format conversion [10], and similar 'lightweight' data transformations.

An important attribute of KStreams is that it can operate independently of applications, as with well-known services like *sendfile* : once initiated, data streaming can proceed without additional user-level input. Toward these ends, application-level knowledge about the structure of streaming data is represented with efficient, kernel-level meta-information about data types, accessible to and used by KStreams handlers. At the same time, when needed, explicit events linking applications with KStreams services can be associated with streams. For instance, for transactional data streams in operational information systems [17], data mirroring and specialization can be changed dynamically by application-level code that assesses current system risks, indicates necessary trust levels [32], or evaluates the current reliability/performance tradeoffs required by applications. Similarly, stream handlers can dynamically interact with web services or sensor data to vary levels of data downsampling, data privacy assurance, and others. Finally, KStreams handlers that implement remote graphical displays, including displays tiled across multiple nodes, KStreams handlers can change their behavior in response to altered end user needs, ranging from rapid changes in a user's viewpoint to slower changes like levels of attention.

The KStreams service is integrated into the Linux operating system kernel above the network protocol and VFS layers. It utilizes the buffer management, flow control and other facilities of the network and file subsystems to acquire and send data. By managing data streams at the kernel level, unnecessary memory copying is avoided, thereby improving system throughput and reducing memory pressure. When operating on 'in-flight' data, stream handlers also require no additional copying, by permitting them to directly operate on data objects read and written from/to network buffers and file caches.

We evaluate the capabilities and performance of KStreams services with microbenchmarks that use different types of incoming and outgoing data objects and multiple ways in which input and output links are utilized. In addition, KStreams is used to implement the functionality needed by applications that include operational information systems, which use transactional data flows, and web or media services that transport visual data. KStreams handlers implement data mirroring, routing, or format conversion functions and/or lightweight image processing actions like data downsampling through image cropping. Initial results are encouraging. Measurements show that in-kernel data streaming can improve system throughput from 20-50% compared to user level implementations, particularly for larger data sizes.

In the remainder of this paper, Section 2 discusses related work, whereas Section 3 describes the basic components of a KStreams data stream. Section 4 describes our Linux-based KStreams prototype and elaborates how this service is integrated into the Linux kernel. Section 5 presents experimental results and discusses the performance of KStreams-based kernel-level data streaming services. Section 6 concludes the paper and presents future work.

2 Related Work

There has been substantial prior work on reducing memory copies in networked systems, much of which was focused on reducing the need to copy data from user to kernel space (or vice versa) for network communications. Basic support is provided by page mapping, where *mmap* , for example, can be used to map the kernel's file cache pages to an application's address space, thereby reducing memory copies on file access. Transparent copy avoidance [21] uses page swapping techniques to implement the copyless interoperation between the network and file subsystems. Its requirement of page alignment for incoming network data limits its applicability, however. Such limitations are addressed by buffer handoff or buffer sharing, as with the Container Shipping I/O system [22]. It uses I/O read and write operations with handoff (move) semantics, which implies that buffers are owned by one domain rather than being shared. In comparison, Fbufs [23] is a copy-free cross-domain transfer and buffering mechanism for I/O data; it uses immutable, concurrently accessible buffers, but Fbufs principally target network communications, so do not support file system accesses. IO-Lite [24] and the zero-copy framework [25] generalize the idea of Fbufs, integrating them with the file cache by implementing a unified I/O buffering and caching system. This solution eliminates data copying via buffer handoff or sharing, but requires that each subsystem's buffer management is changed to utilize the new buffer structure. Our work is less aggressive, by building on top of existing I/O subsystems and not changing the behavior or their respective methods for managing buffers. This implies that data copying will still be required when subsystem boundaries are crossed. It also implies, however, that KStreams is more easily ported to other OS platforms than other approaches, and that our solutions remain open to the additional optimization developed earlier.

A general way to address system throughput is critical path optimization for specific applications. The *sendfile* system call mentioned earlier copies data from file caches directly to the network socket buffers, thereby reducing the number of times data is copied from two (one to applications, one down to network buffers) to one. This is similar to earlier work on splicing data streams [1]. TCP splicing [4] is motivated by web proxying: by 'splicing' two TCP connections, the socket buffer containing incoming data can be directly linked (i.e., re-linked) to the outgoing socket. This involves only small kernel modifications and provides a simple programming interface.

Our work is inspired by TCP splicing, but differs in two ways. First, KStreams allows multiple data objects to act as data sources and sinks, where the relationship among these data objects is defined by application-specific patterns. Second, application-specific stream handlers can be applied to data, where such handlers have full knowledge of the types of data being manipulated. This enables applications to define exactly the data manipulations they need, efficiently apply such manipulations to 'in-flight' data, and control manipulations if and when needed.

Four other kernel facilities developed by our group further enhance the utility of KStreams. First, the PBIO binary data format [18] provides to stream handlers an efficient representation of the application-level data structure contained in message payloads. Second, kernel plugins, realized for IA-32 and IA-64 architectures, permit arbitrary applications to place stream handlers into OS kernels, by ensuring the handlers' name isolation from the remainder of the OS kernel and by ensuring timing safety via time budgets [11]. Third, dynamic binary code generation functionality coupled with a kernel-level, multi-machine resource management infrastructure, termed Q-Fabric [19], lets remote machines place stream handlers into the kernels of machines with which they cooperate, thereby permitting handlers to be deployed on the machines that are best suited to run them. Q-Fabric also provides the efficient event mechanisms needed to permit stream handlers to be dynamically controlled by applications [29]. Finally, a specific class of stream handlers useful for multi-media and real-time applications performs both rate- and deadline-based packet scheduling [14, 20].

3 KStreams Service

Additional data copying limits server throughput by putting pressure on their memory resources. By supporting application-specific data manipulation services at kernel level, the additional data copy required by involving a third party (e.g., an application-level process) in kernel subsystem interactions can be avoided.

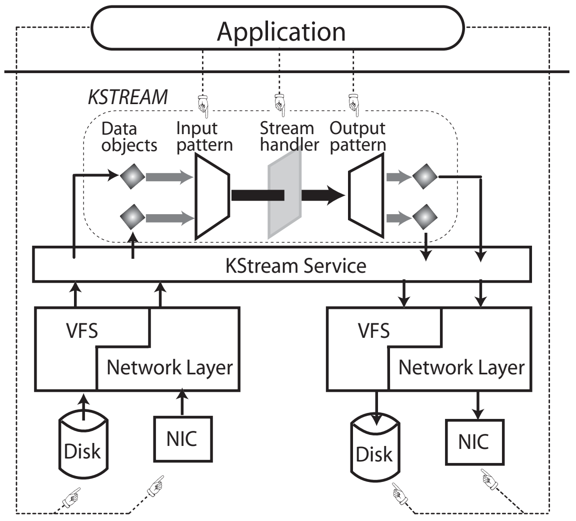

Figure 1: Stream Components.

The challenge is to construct a kernel service that can represent the multiplicity of data transfer models required by applications. Our approach is to support alternative data transfer models by permitting applications to specify: (1) data sources, (2) data sinks, and (3) relationships between sources and sinks. This is illustrated in Figure 1, where arrow lines link application-specified source with sink I/O objects. In addition, 'input patterns' define relationships between multiple sources and 'output patterns' relate multiple sinks. Entire data flows are defined by linking input with output patterns. Finally, application-specific data manipulations are performed by *stream handlers* . Such handlers may change 'in-flight' data, or they may implement value-added services like monitoring, data extraction to detect illicit payload changes, etc.

3.1 Data Sources and Sinks

Since KStreams is implemented as a thin layer on top of native I/O subsystems, it must utilize the virtual file and network systems to provide device-side I/O operational support, cache management, flow control, and interrupt handling. To cleanly interface with different subsystems, the KStreams service defines a data object abstraction that hides the operational details of various devices and subsystems. This abstraction encapsulates the basic operations of I/O devices into a unified interface. KStreams interacts with I/O objects like files and sockets through the data object abstraction in two ways, as shown in Figure 2. First, the data stream can read or write a data object with explicit read/write downcalls. Second, the data object can notify the data stream of important device-side events like data-ready, connection errors etc. via notification upcalls.

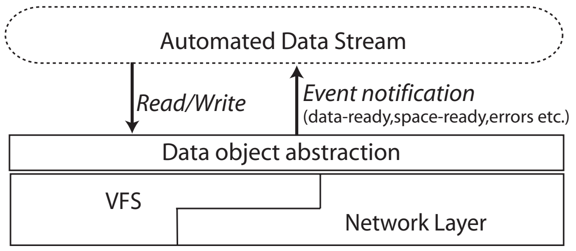

Figure 2: Data Object Interface.

From the application's point of view, data may come from either single or multiple I/O objects. For the latter case, the application should specify how the data coming from different I/O objects is organized. In KStreams, a basic relationship pattern for multiple input points is defined by two factors: (1) polling size, and (2) polling order.

Polling size defines how much data to read from each of the input links every time it is accessed. The size depends on whether the incoming data is a set of fix-sized or variable-sized records. The former case is specified with a single integer, the latter case assumes that size information is included in each of the incoming records. For complex cases, application knowledge is needed to accept incoming data. With KStreams, such knowledge is accessed by allowing the application to specify an in-kernel callback function (one per link) that checks the incoming data and calculates its size. This callback is executed at kernel level, uses meta-information about data structure and sizes provided by applications, and requires some safe method for downloading application code into OS kernels. Several such methods have been developed in past research [2, 30], and the one used in our work is described in [7, 11].

Polling order defines the order by which the data is polled from multiple input data objects. By default, polling order is round robin, but more generally, applications may require multiple inputs to be collected and merged prior to generating an output. In such cases, input patterns may be such that a single record is constructed from portions arriving on multiple links in arbitrary order and/or at different rates. KStreams deals with such application-specific input behaviors by permitting streams to explicitly poll inputs, whenever they arrive or using application-level knowledge accessed via callbacks (e.g., if incoming records are tagged).

Outputs are handled like inputs. An output pattern defines the order in which data is pushed to multiple output data objects, and how much data to write to each of the output links every time it is accessed. As part of the definition, the **pushing order** parameter specifies the manner in which data is written to multiple links (round robin is the default). A simple use of pushing order is data striping, where a single record is divided and striped across multiple output links, perhaps for parallel storage or processing. A more complex example combines such striping with secret sharing techniques in order to confuse potential attackers. Another example is network-aware data output, where data is written to output links according to available link capacities. Multicast pattern is another type of output pattern and can be used for data mirroring and replication.

3.2 Stream Handlers

Stream handlers are useful for many reasons. A sink may require data customization to meet its client's current needs. There may be format mismatches between sources and sinks, due to machine heterogeneity or more generally, due to differences in the assumptions made by end user applications. Stream handlers address issues like these by manipulating data in its fast path from input to output data objects. Stream handlers may be implemented by kernel modules, or dynamically generated as via high level languages or using binary code generation and downloaded into the kernel using isolation support [11].

The effect of placing a stream handler into a KStreams data path is its continuous consumption of incoming and production of outgoing data objects. To attain high performance and avoid additional data copying, such actions must utilize source and sink subsystems' I/O buffer facilities. For instance, logically contiguous source data stored in device-side buffers like file caches and socket buffers is typically scattered across multiple memory buffers. This requires stream handlers to either accept vector-type input buffers or to be implemented to manipulate data incrementally [31]. Both incremental and non-incremental data processing are supported in KStreams, as shown in Figure 3, where non-incremental processing may involve data accumulation in KStreams buffers. As shown later, additional buffering increases memory pressure and reduces system throughput, which indicates that it must be used judiciously.

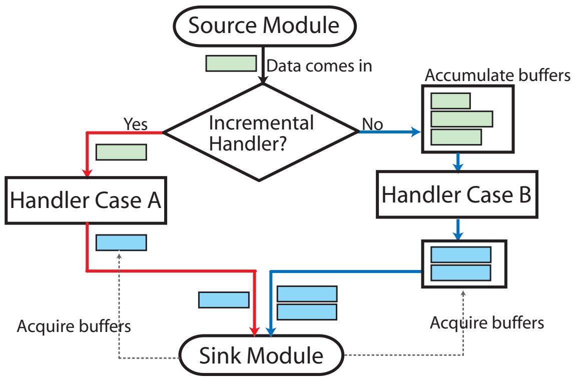

Figure 3: Handler Work Flow.

For 'in-flight' data processing, the fact that stream handlers operate at kernel level enables optimizations that can eliminate additional data copying: since a handler accesses actual I/O buffers via the data object abstraction, processing results can be directly written into and accumulated in sink-side I/O buffers, rather than using additional intermediate buffers. Toward these ends, KStreams provides a simple 'acquire/release' facility for sink-side data buffers.

An illustration of stream handlers placed between source and sink data objects appears in Figure 1. Not shown in that figure is that handlers can also be associated with single, specific output links. The intent is to enable per sink (i.e., per client) data customization.

3.3 Stream Engines

The KStreams service utilizes both synchronous and asynchronous stream engines. A synchronous engine uses an application-level context for execution, thereby guaranteeing that stream handling operates in synchrony with application-level functions. In this case, the application first sets up the KStreams data path, then traps into the kernel via a KStreams system call, and the KStreams service uses the process' context to execute stream handlers and perform I/O operations. The engine determines where to poll for data based on the input pattern, acquires the device-side buffers into which incoming data is placed, provides these buffers to the appropriate stream handlers (if present), determines outgoing data buffers as specified by the output pattern, and finally, writes data to the sink-side I/O buffers. Upon completion or when exceptions occur, control is returned to the application.

The asynchronous streaming model uses a dedicated kernel-level thread to serve multiple streams. One motivation is to support applications that handle a large number of streams, as exemplified by a Web proxy cache application that processes thousands of connections from and to clients and servers. Such a cache typically employs some small number of threads or processes to handle a large number of connections using *select* system call. Another motivation is to provide QoS support. Namely, unlike the single stream model, a kernel-level process can use a QoS scheduler to precisely control resource allocations across multiple streams, without interference from the machine's CPU scheduler.

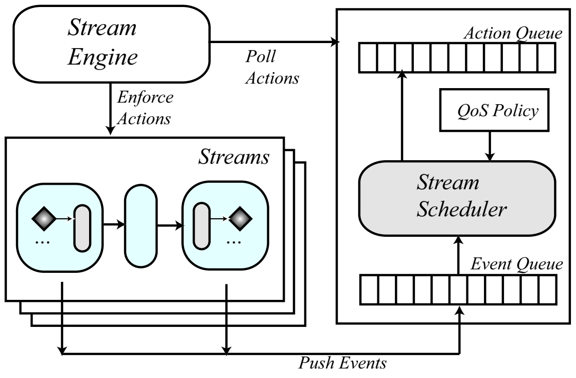

Figure 4: Multiple Stream Control.

An asynchronous stream engine is a kernel-level process driven by control events. Figure 4 depicts an engine that serves multiple streams. Streams' data sources and sinks generate events, typically due to interactions with the underlying I/O subsystems. Three types of important events are *DataReady*, which indicates that there is data available at the source side, *SpaceReady*, which means that there is buffer space available at the sink side, and *ControlEvent*, which are notifications of exceptions, errors, or other special events. Event implementations typically rely on interceptions of subsystem callbacks, as with the *socket* interface, where intercepting the callbacks of a socket in the network subsystem can be used to generate events. Given such events, the stream engine operates by polling actions from the action queue and then enforcing them through the unified data object interface for streams' data sources and sinks.

Events can also be used for QoS control. QoS events are inserted into a KStreams event queue, and a stream scheduler translates such events into associated actions. A best-effort scheduler, for instance, would use a FIFO queue and translate a *DataReady* event to an action that polls for incoming data; a *SpaceReady* event would be interpreted as a data pushing action.

3.4 QoS on Streams

The KStreams service provides quality of service support on multiple streams using a QoS policy-dependent stream scheduler. This scheduler is a replacable element of the KStreams service. It controls resource allocation across multiple streams by issuing I/O operations at different rates for different streams. Since QoS control cannot be limited to single resources, but must address all of the resources used by data streaming actions, QoS scheduling describes the total volume of data being streamed as an explicitly manageable virtual resource. We make this choice for simplicity and ease of understanding, because the total volume of data streaming is closely related to the use of physical resources like CPU, memory, network and disk I/O bandwidth. Finally, since it only controls the rates or data volumnes of data streams, QoS scheduling can delegate to subsystems the allocation of their own system resources.

A simple example of a stream scheduler is a priority based scheduler, where each stream has an associated priority. The scheduler translates events from data sources and sinks into actions, and then sorts the action queue by stream priority. A more interesting example is a class-based fair share scheduler [15], where data streams are classified into several groups, called classes. Classes can be derived from application types, stream importance, or even user IDs. Each class has a share limit that specifies the percentage of system resources allocated to that class of streams. The stream engine executes actions from the action queue in consecutive rounds. In each round, there is a target volume for each class calculated based on its share. The stream engine enforces actions for each class until reaching its target or until there is no action available. The total volume of each round can be varied to ensure reasonable responsiveness and throughput. Unused volume from each class can be partially forwarded to the next round for fairness purposes.

4 Prototype Implementation

KStreams is implemented in Version 2.4.22 of Linux kernel. The current prototype implements data sources and data sinks on regular files and on network sockets. It can use application-specific stream handlers, but does not yet support dynamic handler downloading and runtime handler-application interactions. In the remainder of this section, we comment on some interesting aspects of the KStreams implementation.

Access device-side buffer	***Subscribe for device-side events***
polldata	waitdata
pushvmdata	waitspace
pushagedata	***Callback for event notification***
pushskbdata	dataready
Request device-side buffer	spaceready
getmem	statechange
putmem	

Table 1: Data Object Interface.

4.1 Data Objects

The *data object* abstraction provides a unified interface with which KStreams services access device-side buffers. It also allows services to define specific actions via callback functions in response to device-side events. The API for data objects may be categorized into four groups, as listed in Table 1.

An interesting aspect of the implementation is that I/O buffers acquired via the *polldata* call can be passed to the sink side with three different methods, depending on where data is stored. The first method is used when data is stored as a virtual memory block. It copies data into some sink-side buffer using *pushvmdata*. If data is stored in a physical page and the sink-side buffer is page-based, then *pushpagedata* allows that page to be used as a direct sink-side storage container, protected as 'copy on write'. When data is stored in a socket buffer not associated with any socket, and the sink side link is a socket with same attributes, then *pushskbdata* allows the socket buffer to be directly linked to the new socket. Finally, when output data is generated by stream handlers, the most efficient implementation is one in which a handler writes data directly to a sink side buffer. *Getmem* returns a vector of I/O buffers of some required size, and *putmem* commits the handler's updates to those buffers.

Event subscription and notification callbacks provide an opportunity for KStreams services to respond to special events. This is particularly useful when a single thread operates on multiple streams, since a service cannot block on any single stream. Instead, a service can switch between multiple streams, depending on the availability of incoming data and buffer space, using KStreams events.

4.2 Handler Formats

Stream handlers can operate on stream data as long as they comply with the KStreams API. Typically, a handler consists of two functions, one for data processing and the other for controlling handler execution. Figure 5 depicts the format of handler functions.

```
Interface StreamHandler
{
    int handlercontrol(ctrlcode, param);
    int processdata(inbuffer, insize,
            outbuffer, outsize, state);
    int processdatav(in_iovec,inlen,
            out_iovec, outlen, state);
}
```

Figure 5: Handler Interface.

The function *handlercontrol* is used to initialize handler state, calculate the buffer size for storing result data for given incoming data, and perform similar tasks. A handler either implements *processdata* to process data incrementally, or it implements *processdatav* to process vector type data that is comprised of a fully received, meaningful data record. The state accessible to stream handlers is intentionally limited, defined by a small block of pre-allocated kernel memory for each handler. Such state may contain selected history information or parameters used to determine its current operation.

4.3 KStreams API

The KStreams API for applications is designed to create and control the kernel-level data paths provided by the KStreams facility. The interface addresses path setup, data transfers along the path,

Setup Data Path	**_Transfer Data_**
stream	runstream
attachinput	startstream
detachinput	stopstream
attachoutput	waitstream
detachoutput	**_Stream Options_**
addstreamhandler	setstreamoption

Table 2: KStreams Service Interface.

and path option setting. Table 2 lists the main elements of this API.

The following text describes the typical way in which this API is used. First, the application uses *stream* to create an instance of a data path in the kernel, with polling and pushing orders specified. Next, it attaches files and sockets for input and output by invoking *attachinput* and *attachoutput* for each. Finally, it inserts stream handlers by calling *addstreamhandler*. For synchronous operation, the application uses *runstream* to trap into the kernel and block until an exception occurs or all data is transferred. For asynchronous operation, the application calls *startstream* to delegate the data transfer job to a dedicated thread. The function returns immediately. The application can then *poll* the data path to check on its status, it can terminate the stream by calling *stopstream*, and block on it with the *waitstream* call. In addition, by using *setstreamopt*, the application can set arbitrary key-value pairs for each of the data paths. This API is intended for more complex or for QoS-controlled services.

4.4 Discussion

While the current KStreams implementation can support complex data streaming services like data splicing, data striping, mirroring, and others, some limitations remain. Our current prototype uses pre-created stream handlers loaded as trusted kernel modules, thereby not dealing with the well-known safety and security problems created by running application-level code in OS kernels. We are addressing these issues by integrating the kernel plugins isolation facility described in [11]. Another limitation is related to the multicast pattern for outgoing data, for which we must copy data to the multiple buffers for each outgoing link. We are now investigating changes to the structure of socket buffers to permit one block of buffers to be shared across multiple sockets.

KStreams are designed to efficiently support longer term, large-scale data streaming. Their setup and teardown overheads would not permit their use for short term transfers, like individual http requests, for instance. Further, stream handlers are intended for lightweight data processing like payload monitoring, data filtering, or per-client data customization. Compute-intensive actions like data compression or encryption should be performed at user level, perhaps coupled with KStream-level services that couple such actions with data replication or data striping, for instance.

5 Experimental Evaluation

The performance of KStreams services is evaluated from several perspectives. Microbenchmarks capture the throughput improvements attained by using KStreams vs. application-level data streaming for multiple scenarios. The utility and limitations of kernel-level stream handlers are evaluated with simple sensor/image processing handlers that implement runtime tradeoffs in the amounts of data moved to remote clients. The intent is to emulate the operation of a media data server that customizes data in accordance with current client needs vs. available network bandwidth [28]. A third set of examples are derived from web services applications. Their plug and play capabilities frequently require the dynamic conversion of data formats to match differing sender/receiver assumptions, or to maintain privacy by sharing only the information needed by others [17] (e.g., sharing only low-resolution rather than high-resolution data with subcontractors). The format-aware data streaming services used in our research can provide per-client customized data with high levels of efficiency.

Experiments are conducted on an Intel Pentium III 600MHz machine with 256M memory, connected by a NetGear GA620 Gigabit Ethernet card. Client-side applications run on a much faster 4-way 2.8GHz machine with a Gigabit link, thereby never constituting a bottleneck.

5.1 Microbenchmarks

The first benchmark results evaluate the basic performance of the KStreams infrastructure by experimenting with different kernel-level subsystems, including disk to network, network to disk, and network to network. In addition, multiple stream patterns with respect to the number of incoming or outgoing links are tested. To better understand the implications of 'in-flight' data stream manipulations, a 'dummy handler' outputs exactly the same data as received but also touches each file data entry, thereby emulating a handler that must touch all of a message's payload.

File to socket data streaming is the basic operation in applications like web servers, multimedia services, etc. Figure 6 compares the performance of different implementations of a simple file server. The file server is a multi-thread application that serves file with one thread per request. 16 clients continuously send requests for files of differing sizes. Measurements demonstrate that the KStreams service performs almost as well as the *sendfile* system-call when serving static files. Both the KStreams- and the *sendfile*-based server implementations outperform an implementation that uses read/write system calls. Maximum throughput is limited by network speeds, as evident when file sizes reach 512K bytes. Note that KStreams substantially outperforms user-level implementations that manipulate data with the same 'dummy handlers' as those used at kernel-level, which cannot take advantage of the 'sendfile' command. The user-level implementation is from 30-50% slower than the KStreams implementation, depending on file sizes.

Another typical functionality used by applications is socket to file data streaming, as with data upload, remote backup, disk caches, etc. Using the same file server as above, Figure 7 shows that KStreams-based implementations improve throughput by up to 20% compared to user-level implementations. This is because the user-level implementation must use an additional buffer compared to the KStreams

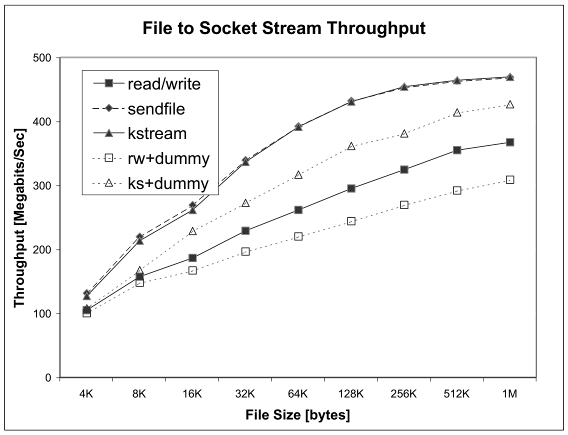

Figure 6: File to Socket Streams.

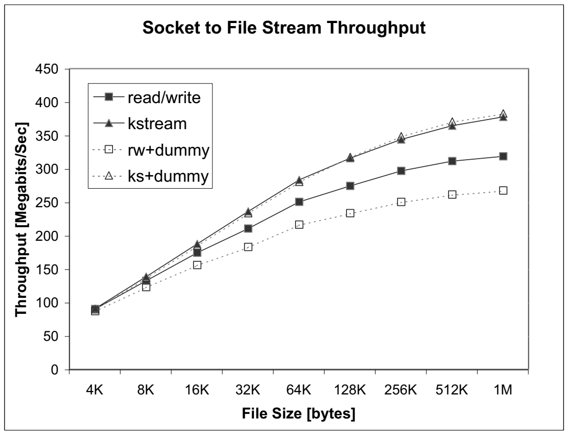

Figure 7: Socket to File Streams.

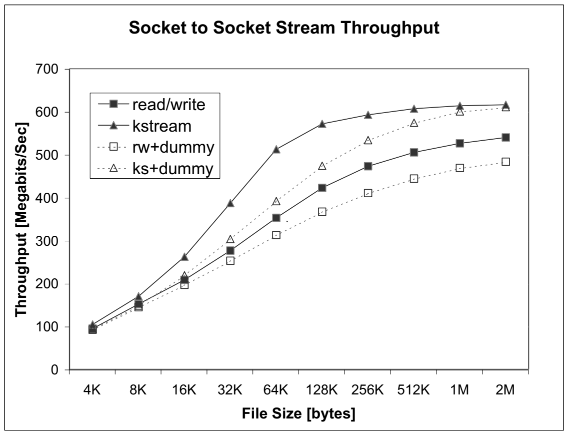

Figure 8: Socket to Socket Streams.

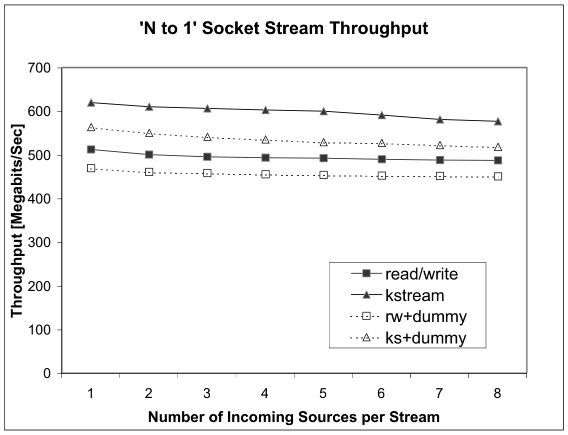

Figure 9: N to 1 Socket Streams.

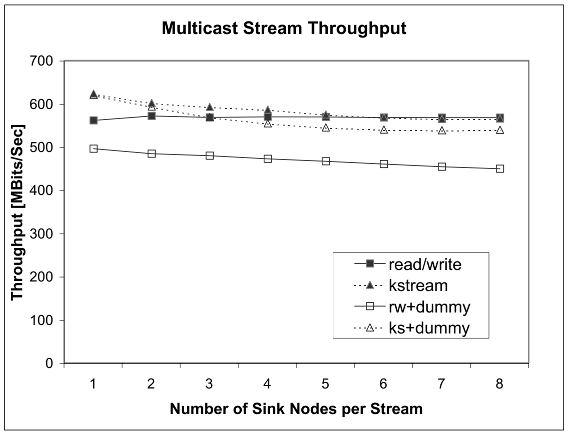

Figure 10: Multicast Streams.

implementation. In addition, adding a dummy handler to the KStreams implementation does not affect total throughput, because the handler can directly operate on the device-side buffer and write to the sink-side buffer. Average improvements are less than those attained for the previous case because the mechanisms of sending to vs. receiving from a network differ.

Socket to socket data streaming is the typical workload of proxy servers. Our experiments use a simple multi-threaded proxy application that relays client requests for files to backend file servers, and forwards the response data back to clients using the KStreams service vs. regular read/write system calls. Figure 8 illustrates the attained throughput, counting both the incoming data from the backend server and the outgoing data to clients. The figure shows that the relative improvement for the KStreams-based service reaches its peak for file sizes around 64K to 128K bytes. When file sizes increase further, throughput improvement become less significant. This is because the network has become the bottleneck. We expect that KStreams-based services will perform even better with large files when using machines that have better network connectivities. Moreover, when dealing with 'smart proxy' applications that can process data 'in-flight' to provide value-added functionality like data monitoring, data filtering, data customization etc., we expect KStreams-based services with dynamic stream handlers to substantially outperform user level implementations. This is evident from measurements that use dummy handlers depicted in Figure 8.

The next set of microbenchmarks concern data streams with multiple incoming and/or outgoing links. Several link patterns are evaluated. For the multiple incoming links, we experiment with data being polled in a round robin manner, in random order, or using self-contained global position tags like timestamps. The performance results shown here use round robin, but we note that other patterns perform similarly.

The specific experiment performed emulates a proxy application that receives data uploaded from multiple sources and then sends the merged data to a backend server, as often done in sensor data fusion. Both incoming and outgoing points are network sockets. The proxy retrieves 4K bytes data per access to each incoming link. The proxy serves 16 streams concurrently, with each stream lasting about 2 minutes. We measure average throughput during data streaming after a 20 second warm-up period, as shown in Figure 9. It is apparent that the throughput attained with KStreams reaches around 600MBits, which is almost the maximum capacity of our Ethernet card and its current driver. It outperforms the user-level implementation by 20%, with better results expected for machines that have stronger network connectivities (i.e., multiple concurrent gigabit links), as is the case with many modern proxy servers.

For the outgoing side, we have experimented with multiple patterns, including round robin for data striping, random for load balancing, and multicast for data mirroring and replication. While the streams with the first two patterns perform similarly to the stream with only one outgoing link, the multicast pattern differs significantly. Figure 10 illustrates the performance of a proxy application that multicasts data incoming from one network link to multiple outgoing links. For this 'data increasing' KStreams service, KStreams plus dummy handler still outperforms the corresponding user level implementation, but for simple data multicast to a small number of nodes, it performs only slightly better. For a larger number of nodes, its performance is almost the same. There are two reasons for this behavior. First, with a larger number of outgoing links, the network becomes the bottleneck. Second, whenever data is copied to multiple outgoing links, it must be copied to multiple buffers, one per link. When multicasting data to N nodes, the KStreams service makes N data copies, whereas the user level implementation makes $N+1$ copies using read/write system calls. If we could modify the socket buffer structure so that multiple sockets could share one piece of data, then the in-kernel data path would be improved further.

5.2 Smart Image Server

Proceedings from microbenchmarks to evaluations that involve more realistic applications, our first application is a 'smart' sensor/image data server used by clients to retrieve PPM-structured data (PPM rather than compressed data is used to permit clients to perform image processing tasks not possible after lossy compression methods are applied). This application evaluates the utility of kernel-level stream handling by downsampling sensor data as per current client needs, using application-specific downsampling methods.

The image server is a multi-threaded application that serves images of PPM format in response to clients' requests. In comparison to a web server that delivers static image files, the application customizes images with client-provided stream handlers. For example, it can grayscale images for use with monochrome display devices, it can crop images or otherwise downsample them to reduce network bandwidth needs and conserve client processing power. The specific handlers used include image cropping, image grayscaling, and three different downsampling methods. The cropping handler chooses one block of the image based on user-specified coordinate values. The grayscale handler just converts each RGB triple to one grayscale value. The three down-sampling handlers use different algorithms. The first one can only down-sample the image to some integral fraction of its original size (e.g., 1/2 or 1/3). The second handler calculates the corresponding neighborhood in the original image for each pixel in the new image, then chooses the middle point of the neighborhood. The third algorithm calculates the neighborhood in original images too, but uses linear interpolation to determine the new color value of the pixel in the new image.

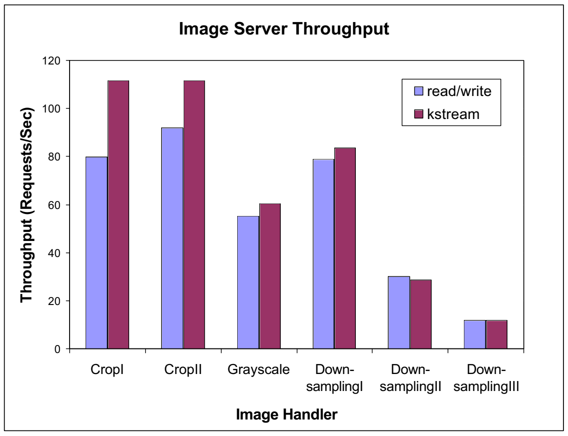

Figure 11: Rquests for Processed Images.

All image handlers process data incrementally. Consider the grayscale handler as an example. The handler uses a small block of pre-allocated space for state variables to store parameters and intermediate processing result. When the PPM image header arrives, the handler parses the image meta-information and stores it as intermediate state. For each triple RGB value, it then computes a grayscale value and outputs it directly to the sink side buffer. When a RGB triple crosses the boundary of an incoming data fragment, the partial value is stored in intermediate state waiting for the next data fragment.

Figure 11 shows the achieved request rates using in-kernel streaming with stream handlers vs. using a user level implementation. Parameters for requests are set so that the output images are approximately one third of the size of the incoming or original images. The exception is 'down-sample I', which produces images of one fourth of their original sizes. The size of the original PPM image is about 900K bytes. Performance improvements are apparent. For CropI, for instance, the KStreams implementation can serve about 40% more requests compared to a simple user level implementation. Crop II is an optimized user-level implementation that uses the *writev* system call, but the KStreams implementation still outperforms it by 20%. In contrast, for computationally expensive image handlers like 'grayscale', performance improvements are less apparent (only about 10%). This trend continues: for down-sampling I, we gain only 6% since computational costs are dominating the savings attained from reducing memory copies; for down-sampling II, and III, the KStreams service performs slightly worse than the user-level service, thereby demonstrating the need to carefully evaluate the data handling performed at kernel vs. user levels.

5.3 Format-Aware Data Services

Most of the format-aware data services implemented in our work operate non-incrementally. This is because dealing with an application-level format typically requires the receipt of an entire application-level message, followed by its manipulation and output. The idea is, of course, that the data server should send data to clients in the formats they require, thereby offloading such manipulations from clients and reducing network bandwidth needs. In our implementation, data records are stored in files that use the PBIO binary data format. When clients send requests for data, they specify the desired format using an XML description, where the desired format may contain only some fields of the server-side data records. The desired format can also require to derive new values for selected entries in new data records, from values contained in the original records. An example is a total passenger count forwarded to a caterer derived from a flight record in the operational information system described in [17]. For such computations, the server uses dynamically generated code to convert old to new data records.

The specific data record used in our experiments is comprised of multiple data fragments received from the source side device buffer. All of these fragments are passed to the handler. The record contains two kinds of fields. The first kind contains integers, floats, etc. This data must be stored in a contiguous memory block. The second kind contains arrays of raw data and strings, where field manipulations can be done even when crossing fragment boundaries. By splittin each record into sub-structures that are either contiguous, thereby requiring an additional copy, or non-contiguous (i.e., not requiring an additional copy), acceptable levels of performance can be

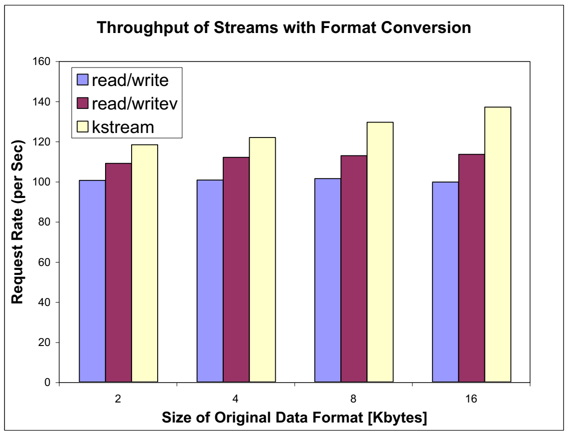

Figure 12: Format Conversion Hander on Streams.

attained for the format and data conversions being implemented, for both source and sink side buffers. The format conversion handler then operates as follows.

Step 1. Translate the data record incoming as multiple fragments into several smaller structures, each of which are placed into a contiguous buffer or represented a array data. Store some small structures in intermediate contiguous buffers, as necessary.
Step 2. Translate the sink side buffers into small structures based on the outgoing data format. Associate small structures with intermediate contiguous buffers if they cross buffer boundaries.
Step 3. Perform format conversions based on small structures.
Step 4. For the converted result, if data is stored in intermediate buffers, move it to the sink buffers.

Because we have not integrated dynamic code generation into the current KStreams implementation, several hardcoded format converters are used. Measurements with these handlers depicted in Figure 12 use 512k byte files that contain data records of 2K, 4K, 8K or 16K bytes. Each data record contains one fourth contiguous (integer or float) and three fourth of array type data. The client requests half of the contiguous data and two third of the array data. The figure shows that the KStreams implementation of format conversion handlers performs better with large data records, and that it outperforms the user level implementation by 38% for 16K bytes records, and by 20% against the optimized implementation using *writev*.

6 Conclusions and Future Work

The contributions of this paper are the definition, implementation, and evaluation of a small set of kernel abstractions that support the efficient implementation of data streaming applications. The KStreams service abstracts files and sockets into unified data objects and links them into an in-kernel fast data path. It defines relationship pattern for complex data streams that consist of multiple incoming or outgoing points, and it permits application-specified data processing to be 'plugged into' the in-kernel fast path, the latter represented as stream handlers. Kernel-level stream scheduling also opens opportunities for QoS control applied to multiple data streams.

Our initial experiences with using KStreams-based services are encouraging. Measurement shows that in-kernel data streaming can improve system throughput from 20-50%. More importantly, in-kernel data streaming reduces the number of times data is copied, thereby reducing the memory pressure imposed on server machines.

Our future work will address multiple issues. First, we are currently integrating the KStreams service with the kernel plugins developed by our group [11], thereby addressing the safety and security problems caused by running user-provided at kernel level. Second, we will investigate the possibility of further optimization of the in-kernel data path. One goal is to allow multiple sockets to share a block of data, thereby enabling efficient kernel-level multicasting. Third, it is known that the application may loses control on the data streaming once it delegates the data transfer tasks to kernel [26]. We plan to integrate different QoS policies with the stream scheduler so that the application can control the behavior of multiple kernel-level data streams.

References

[1] K. Fall and J. Pasquale, Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability, In *Proceedings of the 1993 Winter Usenix Conference*, 1993.

[2] D. Engler, M. Kaashoek and J. O'Toole Jr., Exokernel: An Operating System Architecture for Application-Level Resource Management, In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[3] D. Dias, W. Kish, R. Mukherjee and R. Tewari, A Scalable and Highly Available Web Server, In *the 41st IEEE International Computer Conference*, February, 1996.

[4] D. Maltz and P. Bhagwat, TCP Splicing for Application Layer Proxy Performance, *IBM Research Report RC 21139*, March, 1998.

[5] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, Locality-Aware Request Distribution in Cluster-based Network Servers, In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[6] L. Aversa and A. Bestavros, Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting, In *Proceedings of IPCCC'2000: The IEEE International Performance, Computing, and Communication Conference*, February 2000.

[7] C. Poellabauer, K. Schwan, G. Eisenhauer, and J. Kong, KECHO - Event Communication for Distributed Kernel Services, In *Proceedings of the International Conference on Architecture of Computing Systems*, April 2002.

[8] M. Rosu and D. Rosu, An Evaluation of TCP Splice Benefits in Web Proxy Servers, In *the 11th International World Wide Web Conference*, May, 2002.

[9] A. Gavrilovska, K. Schwan, and V. Oleson, A Practical Approach for 'Zero' Downtime in an Operational Information System, In *22nd International Conference on Distributed Computing Systems*, July 2002.

[10] A. Gavrilovska, K. Mackenzie, K. Schwan, and A. McDonald, Stream Handlers: Application-specific Message Services on Attached Network Processors, In 10th Symposium on High Performance Interconnects, August, 2002.

[11] I. Ganev, K. Schwan, and G. Eisenhauer, Kernel Plugins: When A VM Is Too Much, To appear on *the 3rd Virtual Machine Research and Technology Symposium*, May, 2004.

[12] Linux Source Code, Linux, www.kernel.org

[13] S. Nagar, H. Franke, J. Choi, C. Seetharaman, S. Kaplan, N. Singhvi, V. Kashyap, and M. Kravetz, Class-based Prioritized Resource Control in Linux, In *Poceedings of the Linux Symposium 2003*, July 2003.

[14] R. West and C. Poellabauer, Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams, In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, 2000.

[15] J. Kay and P. Lauder. A fair share scheduler, In *Communications of the ACM*, Jan 1988.

[16] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services, In *Proceedings of the 12th international conference on World Wide Web Conference* , 2003.

[17] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin, Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software*, October 2000.

[18] G. Eisenhauer, Portable Self-Describing Binary Data Streams, *Technical Report GIT-CC-94-45*, College of Computing, Georgia Institute of Technology.

[19] C. Poellabauer, H. Abbasi, and Karsten Schwan, Cooperative Run-time Management of Adaptive Applications and Distributed Resources, In *Proceedings of the 10th ACM Multimedia Conference*, December 2002

[20] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik and R. West, Cooperative Run-time Management of Adaptive Applications and Distributed Resources, In the 4th Real-Time Linux Workshop, December 2002.

[21] J. Brustoloni, Interoperation of copy avoidance in network and file I/O, In *Proceedings of the IEEE Conference on Computer Communications*, April 1999.

[22] J. Pasquale, E. Anderson and P. Muller, Container Shipping: Operating System Support for I/O-Intensive Applications, In *IEEE Computer*, March 1994.

[23] P. Druschel and L. Peterson, Fbufs: A high-bandwidth cross-domain transfer facility, In *Proceeding of the 14th ACM Symposium on Operating System Principles*, December 1993.

[24] V. Pai, P. Druschel and W. Zwaenepoel, IO-Lite: A Unified I/O Buffering and Caching System, In *ACM Transactions on Computer Systems*, February 2000.

[25] M. Thadani and Y. Khalidi, An efficient zero-copy I/O framework for UNIX. *Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc.*, May 1995

[26] J. Kong, D. Rosu, and M. Rosu, Towards Enabling Web Proxy Control of TCP Splice Transfer Rates, In *2nd New York Metro Area Networking Workshop*, September, 2002

[27] C. Poellabauer and K. Schwan, Kernel Support for the Event-based Cooperation of Distributed Resource Managers, In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

[28] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, Cluster-based Scalable Network Services, In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October, 1997.

[29] C. Poellabauer, K. Schwan, and R. West, Lightweight Kernel/User Communication for Real-Time and Multimedia Applications, In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, June 2001.

[30] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers Extensibility, Safety and Performance in the SPIN Operating System, In *Proceedings of the 15th Symposium on Operating Systems Principles*, December, 1995.

[31] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, Building a Robust Software-Based Router Using Network Processors In *Proceedings of the 18th Symposium on Operating Systems Principles*, October, 2001.

[32] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, Responsive Security for Stored Data, In the 23rd International Conference on Distributed Computing Systems, May, 2003